MONITORING ORIENTED PROGRAMMING AND ANALYSIS

BY

FENG CHEN

B.S., Peking University, 1999
M.S., Peking University, 2002

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Doctoral Committee:

Associate Professor Grigore Roşu, Chair
Professor José Meseguer
Professor Lui Sha
Assistant Professor Darko Marinov
Doctor Wolfram Schulte, Microsoft Research

Abstract

This thesis proposes runtime monitoring as a central principle in developing reliable software. Two major research directions are investigated. The first, called *monitoring-oriented programming (MOP)*, aims at detecting and recovering from requirements violations at runtime. In MOP, a user develops specifications together with code; specifications are synthesized into monitors at compile time, and the monitors are then weaved within the application resulting in a system that is aware of its own execution and can correct itself. The second major monitoring-based research direction addressed in this thesis is *predictive runtime analysis (PRA)*, which aims at detecting errors in programs before deployment. In PRA, the program is executed and a causal model is extracted from the observed execution; the causal model is then exhaustively analyzed for potential violations, this way PRA is able to detect errors that did not necessarily occur in the observed execution but that could appear in other executions. MOP and PRA are intrinsically related, both using the same specification formalisms and monitor synthesis algorithms; the difference is that MOP detects errors contemporaneously with their occurrence, and can thus also recover from them, while PRA detects potential errors that can take place in other –unobserved, but causally possible– executions of the system, and thus the system designer can fix them before deployment. Both techniques are sound, in that all reported errors are real.

Two prototype systems have been implemented that prove the feasibility of the proposed techniques. JavaMOP is a monitoring-oriented programming system for Java, which compiles requirements specifications and recovery actions into optimized aspects, which are further weaved within the Java application using off-the-shelf AspectJ compilers. jPredictor is a predictive runtime analysis for concurrent Java applications. It is based on a low-level instrumentation of binary code which, when executed, emits events together with causal information to an external observer. The observer then exhaustively investigates the resulting causal model, detecting all potential concurrency errors witnessed by the observed execution.

To my loving parents Guofan Chen and Yamei Wu,

my brother Jun Chen and my fiancé Gehui Zhang

Acknowledgments to Feng Chen

Due to a tragic event that led to a premature end of Feng Chen's life, this section was written on the 12$^{\text{th}}$ of August 2009 by Feng Chen's Ph.D. adviser, and it is a short acknowledgment to Feng Chen and his amazing accomplishments during his doctoral studies at the University of Illinois at Urbana-Champaign. Feng successfully defended his PhD thesis on 20$^{\text{th}}$ of July 2009 and, with the exception of this unfortunate section, this thesis was completely written and revised by himself.

Both Feng and I started our academic lives at the University of Illinois at Urbana-Champaign seven years ago, in the fall of 2002. Feng had very strong credentials even at that early stage in his career, having worked at the prestigious Bell Laboratories in Beijing and having published papers in top conferences, so I had no doubt that we would have a fruitful collaboration over the years to come. We have together founded and developed a research agenda in formal methods, software engineering and programming languages – an agenda that eventually led to the creation of the Formal Systems Laboratory (FSL). Feng has not only been my dear student, but in time he has become an invaluable colleague and eventually a very good friend, one whose opinion I always asked when important decisions had to be taken in our group.

Feng's seminal work in runtime verification, some of it included in this thesis, serves as a scientific foundation that challenged several research groups around the world. For example, his work on parametric trace slicing and monitoring turned out to lead to systems that can verify their own executions at runtime with a runtime overhead lower than 10%, which significantly outperformed existing similar systems. Also, his work on sliced causality and predictive runtime analysis has lead to systems whose predictive power exceeded by far that of other existing systems. His research accomplishments have been published and presented in the best international conferences. In fact, Feng's Curriculum Vitae speaks by itself: Feng has published more than twenty high quality papers in top conferences such as ICSE, OOPSLA, CAV, TACAS, ASE, SAS, etc. He gave countless

presentations of his work, and was invited to be an intern at Microsoft Research in Redmond several times. Feng's research accomplishments have been rewarded both by colleagues directly interested in his work and by the Department of Computer Science of our university interested in rewarding the best students in all areas: Feng has obtained the ACM SIGSOFT Distinguished Paper Award for his paper in the ASE 2008 conference, and the C.L. and Jane Liu Award offered once a year to a most promising graduate student in the Department of Computer Science of the University of Illinois at Urbana-Champaign, regardless of his or her area.

This tragic event saddens us even deeper when we think of the bright future that Feng had in front of him. Early this year Feng interviewed and accepted a tenure-track assistant professor position in the Department of Computer Science at Iowa State University. After depositing this thesis in mid-August 2009, Feng's plan was to move to Iowa to start his career as a Professor. Moreover, we co-founded a start-up company, targeted at further developing and eventually commercializing the technologies initiated by Feng during his doctoral studies. His professorship position at a very good university, together with his software company co-founder status, put Feng in an elite category of computer scientists, comprising no more than a handful of such distinguished professionals.

Feng's sudden death has generated shock-waves both in the Department of Computer Science at the University of Illinois and in the scientific community. Uncountably many messages have been received from colleagues and friends all over the world, expressing their sorrows for the unbelievable loss. Nevertheless, no matter how big the loss of Feng is to the scientific community and his friends, it cannot be compared to the loss of a son, a brother, and a fiancè. Feng's numerous colleagues, professors, friends, and myself, convey our condolences to Feng's father Guofan Chen, to Feng's mother Yamei Wu, to Feng's brother Jun Chen, and to Feng's fiancé Gehui Zhang. Feng was highly regarded and he will be deeply missed by all of us. May his soul rest in peace.

Grigore Roşu
Associate Professor
Department of Computer Science
University of Illinois at Urbana-Champaign

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As we are in a society increasingly dependent on software, the consequences of programming errors are becoming increasingly important. Significant material values, sometimes even our lives, may depend on the correct behavior of software. The discovery and prevention of software errors has therefore become an extremely important yet very difficult problem involving many aspects, such as incorrect or incomplete specifications, errors in coding, or faults and failures in hardware or operating system. This thesis addresses two related aspects of the problem, namely, checking that the behavior of a system meets its specification and modifying the system's behavior if its specification is violated. Our research results in a systematic and tool-supported software development methodology that regards runtime monitoring as a basic development principle to improve software reliability.

Many engineering disciplines consider and accept monitoring as a major design principle to increase safety, reliability and dependability of their products, such as fuses and watchdogs. Similarly, runtime monitoring of requirements in software development can increase the reliability of the resulting software systems. On the one hand, if monitoring is used as integral part of a system to *detect and recover from requirements violations* at runtime, then monitoring can increase the dependability and safety of the deployed system, by guiding the running system to avoid catastrophic failures; in aircraft and spacecraft systems, for example, automatic, rather fancy controllers are typically monitored to ensure that their predicted state stays within a "stability envelope", from where the system can always be safely controlled in a timely manner using slower but better understood and safer procedures. On the other hand, if used to detect errors in programs before deployment, monitoring can bring more rigor and more power to testing. Monitors can verify not only states of programs at specific points, but also temporal behaviors referring to complex patterns and histories of actions. Monitors can even *predict* potential behaviors of the monitored program, which may have not occurred, from what have been observed, significantly increasing the coverage of testing.

We have developed a generic and automated framework for application of runtime monitoring in software development, called *monitoring-oriented programming* (MOP). MOP aims at reducing

the gap between formal specification and implementation by integrating the two and allowing them together to form a system. Monitors are automatically synthesized from formal specifications and integrated at appropriate places in the program, according to user-configurable attributes. Moreover, specification and implementation can and should *interact* with each other *by design*, rather than by grafting monitoring requirements on an existing system to increase its safety. In MOP, runtime violations and/or validations of specifications can result in adding functionality by executing any user-defined code at any user-defined places in the program. For example, outputting debugging information and/or executing recovery code when the specification is violated is very important, but in our framework it is just one specific use of monitoring among others.

In MOP, one specifies requirements specification using *extensible* specification formalisms. Practice has shown that there is no "silver bullet" logic to express all requirements. Some can be best expressed using a certain logical formalism, for example temporal logics, while others can be best expressed using other logics, like that of regular expressions. For these reasons, the MOP framework provides the capability of adding new monitoring logics through an extensible logic framework. More precisely, as part of our MOP methodology, we introduced the general concept of a *logic plug-in*, as a formalization of the informal notion of "monitoring logic", i.e., specification formalism. Every logic plug-in encapsulates a monitor synthesis algorithm for a particular logic. The interface of logic plug-ins is standardized, allowing one to extend MOP with her/his own logics. A set of logic plug-ins have been created and developed within the MOP framework, supporting a broad range of monitoring logics from finite state machines (FSM) to context-free grammars (CFG).

In practice, many requirements specifications are unavoidably *parametric*. Parametric specifications are specifications with free variables, i.e., parameters, which are instantiated to concrete values at runtime. Efficient monitoring of parametric specifications is highly challenging due to the fact that the connection among parameters can be sophisticated and the number of parameter instances created during an execution can be tremendous. To address this problem, we have developed a *logic-independent and efficient* solution in MOP for monitoring parametric properties. Our solution maintains a monitor instance for every parameter instance encountered at runtime and uses the parameter information as the indexing mechanism to search for corresponding monitor instances. Every monitor instance is regarded as a black box that checks its own trace which contains only relevant events. The MOP framework assumes no knowledge about internal details of the monitor and the monitor does not need to handle parameters unless it chooses to. This way, parameter handling is separated from actual property verification, allowing us to develop a framework that is

not only optimized for monitoring parameter properties but also generic in specification formalisms.

Shortly, one can understand MOP from at least three perspectives:

1. As a discipline allowing one to *improve safety, reliability and dependability of a system by monitoring* its requirements against its implementation at runtime;

2. As an *extension of programming languages with logics*. One can add logical statements anywhere in the program, referring to past or future states of the program. These statements are like any other programming language boolean expressions, so they give the user a maximum of flexibility on how to use them: to terminate the program, guide its execution, recover from a bad state, add new functionality, throw exceptions, etc.;

3. As a *lightweight formal method*. While firmly based on logical formalisms and mathematical techniques, MOP's purpose is not program verification. Instead, the idea is to avoid verifying an implementation before operation, by *not letting it go wrong* at runtime.

MOP can be used to monitor and verify program executions in software testing. Our evaluation has shown that MOP provides an effective solution to detect semantics-related errors that are usually omitted by ordinary software testing (Section 5.3.2). However, MOP's capability of finding errors is limited by the coverage of the underlying testing technique, since MOP checks only what have occurred in the testing process, just like any other monitoring-based approaches. This limitation is especially critical for concurrency-related errors. Many, if not most, real-world software systems are concurrent. Concurrent systems may exhibit different behaviors due to different thread/process interleavings when executed at different times, even with the same input. This inherent nondeterminism makes concurrent programs very difficult to analyze, test and debug. To address this problem, we have developed *predictive runtime analysis*, a technique to infer potential behaviors of a concurrent system from its executions that have been observed. In other words, predictive runtime analysis allows us to see beyond what have occurred and to predict hidden concurrent errors in the monitored system even before the errors have actually happened.

Predictive runtime analysis extracts from an observed execution trace a *causal partial-order dependence* relation and then generates all the consistent linearizations of that partial-order, which have been proven to be valid executions of the observed concurrent system even if they have yet occurred. The inferred executions can be checked against requirements specification, which can be either generic, such as race conditions, or program specific, e.g., user-specified properties, using any existing monitoring techniques, such as MOP. Any violation of the given specification by some

inferred execution reveals a potential error in the system. In other words, by observing an execution trace that may not violate the requirements, one can correctly predict faulty execution traces *without the need to run the program again.*

The causal partial order extracted from the observed execution determines the prediction capability of predictive runtime analysis. The more relaxed is the partial order, the more linearizations can be generated, meaning that more potential executions can be inferred and thus better prediction capability can be achieved. Based on this observation, we have proposed *sliced causality*, a loose "happen-before" causality. Based on an apriori static dependence analysis, sliced causality drastically cuts the causal partial order extracted from the observed execution by removing unnecessary dependencies; this way, a significantly larger number of consistent runs can be inferred and thus analyzed. Sliced causality does not sacrifice *soundness* of the analysis results, i.e., it does not report any false positives, for the increased analysis coverage: every linearization computed using sliced causality is provn to be a feasible execution of the concurrent system under analysis.

We have developed an MOP tool for Java programs, called JavaMOP. JavaMOP instantiates the MOP framework and provides different interfaces to the user, namely, a web-based interface, a Eclipse-based graphic interface, and a command-line interface. Many monitoring logics are supported in JavaMOP based on the extensible MOP logic framework, including linear temporal logics (LTL), extended regular expressions (ERE), and context-free grammars (CFG). We have also implemented jPredictor, a predictive runtime analysis tool for Java. jPredictor combines sliced causality with *lock atomicity*, which captures the semantics of locking mechanisms in Java, providing enhanced predication capability for finding errors in multi-threaded Java programs. Both tools have been applied to many real-world Java programs and detected tricky bugs in popular open-source applications, outperforming other similar tools in both effectiveness and efficiency.

Outline. This thesis is organized as follows. Chapter 2 gives an brief overview of approaches presented in this thesis using a real-world example. Chapter 3 discusses the underlying philosophy of MOP, including the MOP monitoring model and the extensible logic framework. Chapter 4 focuses on the logic-independent framework provided by MOP for efficiently monitoring parametric properties. Chapter 5 discusses the implementation and evaluation of JavaMOP. Chapter 6 introduces predictive runtime analysis and defines sliced causality. Chapter 7 presents jPredictor and evaluates its effectiveness in practice. Chapter 8 discusses related work and Chapter 9 concludes.

Chapter 2

Overview

Here we give an overview of approaches presented in this thesis using a safety example in the Java Util library. We first show how one can specify and check the desired safety property using the MOP framework. Then we discuss the example in the context of concurrent programs, illustrating how to predict potential violations of the property even when the observed execution did not violate the property.

2.1 Monitoring-Oriented Programming

Every practical programming language provides libraries that contains fundamental and important functions shared by different programs, e.g., input/output and basic data-structures. Proper usage of the libraries is critical for developing reliable programs and often requires to follow certain *contracts* of the libraries, e.g., certain orders or patterns of calling related library functions. The library contracts are usually described informally in the library specification, but some of them are considered so important that language designers feel it appropriate to include corresponding runtime safety checks as built-in part of programming languages. For example, a Java virtual machine (JVM) will raise a `ConcurrentModificationException` when running the following piece of code, in which `Vector` is a Java library class encoding the data-structure for vectors and `Iterator` is a Java interface that can be used to enumerate a set of elements:

```
Vector v = new Vector();
v.add(new Integer(10));
Iterator i = v.iterator();
v.add(new Integer(20));
System.out.println(i.next());
```

The first line of code creates a new `Vector` object, `v`, and the second line adds a new element into `v`. The third line creates an `Iterator` object, `i`, for `v`; this way, one can enumerate the elements in `v`

using `i`. In the fourth line of code, another element is added into `v`. At the end, in the fifth line, the program tries to print out the next available element in `i`, which will trigger a runtime exception as mentioned above. That is because any `Iterator` object returned by `Vector`'s `iterator()` method is assumed *fail-fast* in Java: the underlying vector is not allowed to be modified while some of its iterators access its elements, or a `ConcurrentModificationException` will be thrown.

The fail-fast property captures unsafe usage of `Iterator` at runtime and prevents other problems that may be caused by the unsafe usage. But another similar interface provided by Java, named `Enumeration`, which is also used to enumerate elements and can be obtained by `Vector`'s `elements()` method, is *not* assumed fail-fast, and, obviously, neither are any other user-defined iterator-like objects. The lack of support for the fail-fast `Enumeration` may lead to tricky problems in practice: we have detected a bug in an open source application, jHotDraw [55], caused by a violation of the fail-fast property for `Enumeration` using the MOP approach discussed below. Since no exception was thrown when an `Enumeration` object was used unsafely, the program generated unexpected output at a point that is far away from the unsafe usage of the `Enumeration` object, making it difficult to locate the real cause of the problem using ordinary testing techniques.

It is not trivial for a programmer to manually implement the runtime checking of the fail-fast property efficiently and correctly because it involves interaction between two objects and also needs to avoid problems caused by concurrency. Moreover, since `Enumeration` is an interface, every its concrete implementation needs to implement the runtime checking if one wants to enforce the fail-fast property for all `Enumeration` objects. We next show that using an MOP tool, one can have the fail-fast runtime checking for `Enumeration` automatically generated and integrated into any program using `Enumeration`, avoiding all the complexity of manual implementation. Also, the automatic monitor generation and integration provides *stronger guaranty for correctness* with regard to the checked property *without sacrificing efficiency*: our evaluation shows that the monitoring code generated by MOP is as efficient as manually optimized implementation and better than other existing runtime verification systems in most cases (Section 5.3).

In MOP, one first needs to formally specify the desired property using some specification formalism. Figure 2.1 gives the JavaMOP specification for the fail-fast *Enumeration*. A complete and rigorous syntactic definition of JavaMOP specifications is given in Section 5.1. We here only explain the above specification informally. The first line in the specification in Figure 2.1 names the specified property, `UnsafeEnum` in this case, and states that this property uses two parameters: `Vector v` and `Enumeration+ e`; the "+" means that this property (and its monitors) is inherited by all the

```
UnsafeEnum(Vector v, Enumeration+ e) {
        event create after(Vector v)
                returning(Enumeration+ e) :
                call(Enumeration Vector+.elements())
             && target(v) {}
        event updatesource after(Vector v) :
                (call(* Vector+.remove*(..))
             || call(* Vector+.add*(..))
             || call(* Vector+.clear(..))
             || call(* Vector+.insertElementAt(..))
             || call(* Vector+.set*(..))
             || call(* Vector+.retainAll(..)))
             && target(v){}
        event next before(Enumeration+ e) :
                call(* Enumeration+.nextElement())
             && target(e){}
        ere : create next* updatesource updatesource* next
        @match {
            System.out.println("improper enumeration usage at " + __LOC);
        }
}
```

Figure 2.1: Property UnsafeEnum in JavaMOP

subclasses of `Enumeration`. Three events invovled in the property are then defined using AspectJ-like syntax [58]. Event `create` is issued whenever an `Enumeration` object is created for a `Vector` object. It contains two parameter, namely, `v` for the target vector and `e` for the created enumeration. Event `updatesource` is issued when methods modifying a `Vector` are called with one parameter, namely, `v` for the target vector. Event `next` is issued when the `nextElement()` method is called on an `Enumeration` object and uses the parameter `e` for the target enumeration. After the event definitions, an ERE formula, essentially a regular expression in this example, is given to express the fail-fast pattern: at least one `updatesource` event is seen after the `create` event and before some `next` event. Note that events in the formula are assumed parameterized as above. Therefore, the specified formula should be interpreted as: for any pair of `Vector v` and `Enumeration e`, we have that `e` is created for `v` and then `v` is updated, after which `e` is used to access the elements in `v`.

A match of the specified ERE pattern indicates a violation of the fail-fast property. Hence, in the specification in Figure 2.1, we associate a `@match` handler with the specified pattern, which will be executed when the pattern is matched. In this example, the `@match` handler simply reports the line of code at which the pattern is matched, i.e., where an `Enumeration` object is used after the underlying vector has been changed. A JavaMOP reserved keyword `__LOC` is used in this handler, which gives the line of code at which the latest event is issued.

JavaMOP will translate the given specification into AspectJ code. One can use any off-the-shelf AspectJ compiler to weave the generated monitoring code into any program using `Enumeration` and `Vector` to enforce the fail-fast property for `Enumeration`. The generated monitoring code will create many monitors at runtime, each for a corresponding pair of `v` and `e` generated during the execution of the program, and will dispatch the events correspondingly; for example, if several enumerations are created for the same `Vector` object `v`, then an `updatesource` event with parameter `v` is sent to each monitor corresponding to each enumeration of `v`. The monitor will check the received events against the specified pattern and execute the associated `@match` handler when the pattern is matched. The process of creating and locating monitors at a received event can be sophisticated, considering all the possible parameter combinations of events. For example, JavaMOP generates about 200 lines of AspectJ code from the specification in Figure 2.1, which can be nontrivial if implemented manually. More details about the monitoring process are discussed in Chapter 4.

2.2 Predictive Runtime Analysis

MOP provides an effective solution to detect errors that actually occur during an execution. But it suffers from the same limited coverage as testing with regards to error detection. In other words, if the observed execution does not violate the specified property, the monitor generated by MOP will not be able to find any error even when there is a bug in the monitored program. This limitation is especially critical for detecting concurrent errors. Let us consider the example in Figure 2.2.

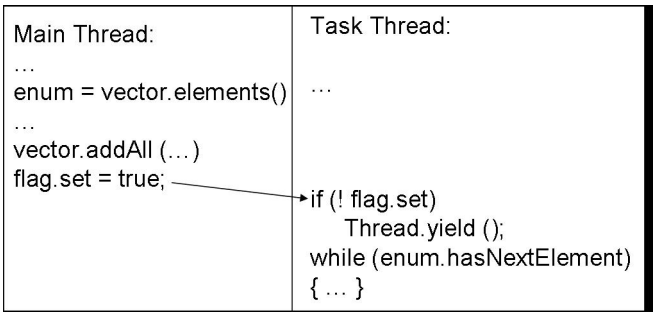

Figure 2.2: Multi-threaded execution

There are two threads in the program in Figure 2.2, namely, the main thread and the task thread. Three variables are used: `vector` is a `Vector` object, `enum` is an `Enumeration` object created for `vector`, and `flag` is used for synchronization between the main thread and the task thread. All three variables are shared between the two threads. The main thread changes `vector` and the task thread uses `enum`. Synchronization is unnecessary, since only the main thread modifies `flag`. This

program is buggy because one of its execution may match the `UnsafeEnum` property discussed above. It is because the developer makes a (rather common [39]) mistake, using `if` instead of `while` in the task thread. Suppose now that we observed a successful run of the program, as shown by the arrow, which has much higher probability to happen during testing than the buggy execution. The monitor generated from the JavaMOP specification in Figure 2.1 will not find the bug because the observed execution does not match the specified pattern in this particular run. However, with the predictive runtime analysis technique that we have developed, one will be able to predict the bug from the observed, successful execution *without the need to re-run the program* to actually hit the bug.

Our approach was inspired by the happen-before causality proposed in [63]. Several approaches have been introduced to detect concurrency bugs based on happen-before techniques, e.g., [72, 81, 82]. They extract causal partial orders from analyzing *exclusively* the dynamic thread communication in executions. But as discussed in [83], without additional information about the structure of the program that generated the event trace, the least restrictive causal partial order that an observer can extract is the one in which each write event of a shared variable precedes all the corresponding subsequent read events and which is a total order on the events generated by each thread. Since this causality considers *all* interactions among threads, e.g., all reads/writes of shared variables, the obtained causal partial orders are rather *restrictive*, or *rigid*, in the sense of allowing a reduced number of linearizations and thus of errors that can be detected; in general, the larger the causality (as a binary relation) the fewer linearizations it has. Hence, those happen-before based approaches provide limited capability of predicting concurrent bugs. For example, they will not be able to find the bug in Figure 2.2, due to the causality induced by the read/write of `flag`.

We have defined *sliced causality*, a causal partial order relation significantly reducing the size of the computed causality without giving up soundness or genericity of properties to check: it works with any *monitorable* (safety) properties, including regular patterns, temporal assertions, data-races, atomicity, etc. Based on sliced causality, we have developed the predictive runtime analysis technique that is more effective and more accurate than other existing techniques. Let us use the example in Figure 2.2 to intuitively explain our approach. In the program in Figure 2.2, the use of `enum` is *not* controlled by the preceeding `if` statement in the task thread, since the `while` loop will executed no matter what choice is made at the `if` statement[1]. Therefore, our technique will ignore the causality induced by the write/read on `flag` because it does not affect the events relevant to the `UnsafeEnum` propety (updates of `vector` and uses of `enum`). Consequently, the

[1]It is more complicated to decide the control dependence when more control flow statements, e.g., exception throwing, are considered. Interested readers can refer to [32] for details.

resulting sliced causality extracted from the observed execution does not impose any order between `vector.addAll()` and `enum.hasNextElement()`. This means that these two statements can be executed in any order, including one matching the UnsafeEnum pattern. The bug is predicted. When the bug is fixed by replacing `if` with `while` in the task thread, the `while` loop on `enum` is controlled by the `while` loop on `flag` (since it is a potentially non-terminating loop). Our technique then will take the causality induced by the write/read on `flag` into account, resulting in a causal order between `vector.addAll()` and `enum.hasNextElement()`. No violation will be reported in this case.

The sliced causality is constructed by making use of dependence information obtained both statically and dynamically. Briefly, instead of computing the causal partial order on all the observed events like in the traditional happen-before based approaches, our approach first slices the trace according to the desired property and then computes the causal partial order on the achieved slice; the slice contains all the *property events*, i.e., events relevant to the property, as well as all the *relevant events*, i.e., events upon which the property events depend, directly or indirectly. This way, irrelevant causality on events is trimmed without breaking the soundness of the approach, allowing more permutations of relevant events to be analyzed and resulting in better coverage of the analysis.

In short, based on an apriori static analysis, sliced causality drastically cuts the usual happen-before causality by removing unnecessary dependencies. It thus allows for a significantly larger number of consistent runs to be inferred and thus analyzed. Experiments show that, on average, the sliced causality relation has 50% or less direct inter-thread causal dependencies compared to happen-before [30]. Since the number of linearizations of a partial order tends to be exponential with the size of the *complement* of the partial order (as a binary relation), any linear reduction in size of the sliced causality compared to traditional happen-before, is expected to *increase exponentially the coverage* of the analysis. Indeed, the use of sliced causality allowed us to detect concurrency errors that are unlikely be detected using conventional happen-before causalities.

Chapter 3

The MOP Philosophy

Monitoring oriented programming (MOP) is a generic framework for runtime verification. Runtime verification (RV) [48, 84, 16] aims at combining testing with formal methods in a mutually beneficial way. The idea underlying runtime verification is that system requirements specifications, typically formal and referring to temporal behaviors and histories of events or actions, are rigorously checked at runtime against *the current* execution of the program, rather than statically against all hypothetical executions. If used for bug detection, runtime verification gives a rigorous means to state and test complex temporal requirements, and is particularly appealing when combined with test case generation [8] or with steering of programs [60]. A large number of runtime verification techniques, algorithms, formalisms, and tools such as Tracematches [3], PQL [66], PTQL [44], MOP [25], Hawk/Eraser [34], MAC [60], PaX [47], etc., have been and are still being developed, showing that runtime verification is increasingly adopted not only by formal methods communities, but also by programming language designers and software engineers.

Most RV approaches focus on particular application domains to achieve better effectiveness and/or efficiency, bound to specific programming languages and specification formalisms (see Section 8.1 for more discussion), despite the fact that they share many fundamental issues, including how to observe the program execution and how to check the observed execution. We focus on genericity in MOP, aiming at providing fundamental support for runtime monitoring instead of a domain-specific solution. More precisely, MOP is generic in both programming languages and specifications formalisms, thanks to its well-defined monitoring model and carefully designed architecture. One can easily instantiate it to fit in different applications domains, greatly facilitating application of runtime verification in practice.

In the rest of this chapter, we first explain the monitoring model and architecture of MOP and then introduce its extensible logic repository. At the end, a generic and efficient parametric monitoring mechanism is presented.

3.1 MOP Monitoring Model

Many properties can be monitored at the same time in MOP. The execution trace against which the various properties are checked is extracted from the running program as a sequence of events taking state snapshots. Events produce sufficient information about the concrete program state in order for the monitors to correctly check their properties. A monitor is typically interested in a subset of events. Figure 3.1 illustrates the monitoring model adopted by MOP.

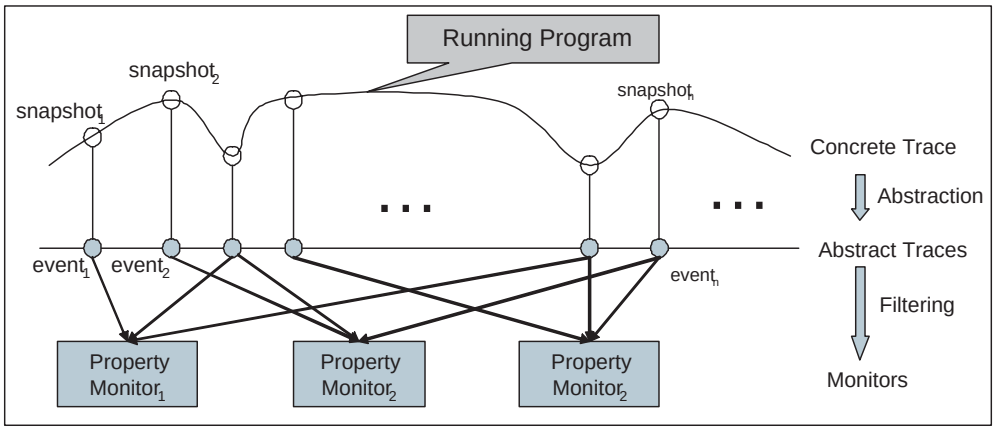

Figure 3.1: MOP Monitoring Model

In MOP, the runtime monitoring process of each property consists of two orthogonal mechanisms: *observation* and *verification*. The observation mechanism extracts property-relevant and filtered system states at designated points, e.g., when property-specific events happen. The verification mechanism checks the obtained abstract trace against the (monitor corresponding to the) property and triggers desired actions in case of violations or validations. For instance, for the simple global property "always $(x > 0)$", the events to observe are the updates of the variable `x` and the relevant state information (or snapshot) to extract is the value of `x`. This observation process yields a sequence of values of x: the relevant abstract trace. The corresponding monitor checks whether the value of `x` is larger than zero. Observation and verification are therefore independent: the running mode of the monitor and/or the algorithm used within the monitor does not affect how the execution is observed, and vice versa. This clear separation in the MOP monitoring model results a highly configurable and extensible monitoring framework, as discussed in what follows.

3.2 Configurable Monitoring

The monitoring model in Figure 3.1 makes no assumption or restriction on the way that the monitor works. This makes MOP a highly configurable and extensible runtime verification framework. A monitor can be used in different ways and can interact with the monitored program at different places, depending on specific needs of the application under consideration. For example, in some applications, the monitor may need to use the same resources as the rest of the program, in others one may want to run the monitors as different processes. The user should be allowed to configure the monitor for her/his needs. MOP thus provides several options that one can use to configure a monitor for different requirements. We next discuss each of them in depth. Note that these options are general, that is, they are not specific to a particular application. An MOP instance may choose to implement some or all of them according to different requirements; also, an MOP instance provide additional configuration options based on its application domain. Some related discussion can be found in Sections 5.1 and 5.2.

Inline or outline, online or offline.

Depending on where the monitoring code is executed, one can distinguish between *inline* monitoring and *outline* monitoring. Under inline monitoring, the monitor runs as a embeded component in the monitore program, using the same resource space as the program. The monitor is usually inserted as one or more pieces of code into the monitored program. In the outline mode, the monitoring code is executed as a different process, potentially on a different machine or CPU. The in-line monitor can often be more efficient because it does not need inter-process communication and can take advantage of compiler optimizations. However, an inline monitor cannot detect whether the program deadlocks or stops unexpectedly. Outline monitoring has the advantage that it allows, but does not enforce, a centralized computation model, that is, one monitor server can be used to monitor multiple programs. In order to reduce the runtime overhead of out-line monitoring in certain applications, one can define communication strategies; for example, the monitored program can send out the concrete values of the relevant variables and the outline monitor evaluates the state predicates, or, alternatively, the monitored program sends out directly the boolean values of the state predicates.

Depending on when the desired property is checked against the observed execution, one can distinguish between *online* monitoring and *offline* monitoring. Under online monitoring, which is the most used case in runtime verification, specifications are checked against the execution of the

program dynamically and run-time actions are taken as the specifications are violated or validated. The off-line mode is mostly used for debugging purposes: the program is instrumented to log an appropriate execution trace in a user-specified file and a program is generated which can analyze the execution trace. The advantage of off-line monitoring is that the monitor has random access to the execution trace. Indeed, there are common logics for which online monitoring is exponential while offline monitoring is linear [76].

Note that the above two pairs of running mode options are orthogonal to each other. In particular, an inline offline configuration inserts instrumentation code into the original program for generating and logging the relevant states as the program executes. Alternatively, an outline offline configuration generates an observer process which receives events from the running program and generates and then logs the states relevant to the corresponding specification. The latter may be desirable, for example, when specifications involve many atomic predicates based on a relatively small number of program variables; in this case, the runtime overhead may be significantly reduced if the program is instrumented to just send those variables to the observer and let the latter evaluate the predicates.

Synchronous monitors.

This option states whether the execution of the monitor should block the execution of the monitored program or not. For critical properties, synchronous monitors are preferred so that actions can be carried out before the detected violations cause actual problems. If the properties under verification do not require immediate actions or take a long time verifiy, one may use asynchrounous monitors which allow the monitored program to contiue when the properties are checked. There can be different ways to implement this option. In in-line monitoring, for example, a synchronous monitor may be executed within the same thread as the surrounding code, while an asynchronous one may create a new execution thread and thus reduce the runtime overhead on multi-threaded/multi-processor platforms.

Synchronous monitors should be distinguished from synchronized monitors. The latter is related to concurrent programs and specific to the underlying programming language: in a concurrent program, different monitors may run in different threads or processes and share certain resources; in such case, synchronized monitors can be used to avoid races on the resources shared by monitors. Synchronicity also plays a role in synthesizing code from logic formulae, because, for some logics, asynchronous monitoring is more efficient than synchronous monitoring. Consider, for example, the

temporal formula "next F and next not F" which is obviously not satisfiable: a synchronous monitor must report a violation right away, while an asynchronous one can wait one more event, derive the formula to "F and not F", and then easily detect the violation by boolean simplification. Note that synchronous monitoring requires running a satisfiability test, which for most logics is very expensive (PSPACE-complete or worse).

Suffix matching.

According to application requirements, one may want to check the desired property against either *the whole execution trace* or *every suffix of a trace*. Total matching has been adopted by many runtime verification approaches to detect pattern failures of properties, e.g., JPaX [47] and JavaMaC [61]. Suffix matching has been used mainly by monitoring approaches that aim to find pattern matches of properties, e.g., Tracematches [13]. PQL has a skip semantics, wherein a specification is matched against the trace, but events may be skipped. A precise explanation of PQL's semantics is available in [66]. To define suffix and total matching we first must define traces and properties:

Definition 1 *Let $\mathcal{E}$ be a set of events. An* $\mathcal{E}$-trace*, or simply a* trace *when $\mathcal{E}$ is understood from context, is any finite sequence of events in $\mathcal{E}$, that is, an element in $\mathcal{E}^*$.*

In the context of monitoring, an execution trace is a sequence of events observed up to the current moment, thus execution traces are always finite.

Definition 2 *An* $\mathcal{E}$-property *P, or simply a* property*, is a pair of disjoint sets (P_+, P_-) in $\mathcal{E}^* \times \mathcal{E}^*$; P_+ is the set of* pattern matching traces *and P_- is its set of* pattern failing traces[1].

Therefore, our notation of property is quite general; for each particular specification formalism, one need associate an appropriate property to each formula or pattern in that formalism. Total matching and suffix matching are then defined as follows:

Definition 3 *The* total matching semantics *of P is a function*

$$[\![P]\!]_{total} : \mathcal{E}^* \to \{pattern\ match, pattern\ fail, ?\}$$

[1] We generalize this concept to generic categories beyond just match and fail in Chapter 4. However, match and fail are sufficient for the discussion.

defined as follows for each $w \in \mathcal{E}^*$*:*

$$[\![P]\!]_{total}(w) = \begin{cases} \textit{pattern match} & \textit{if } w \in P_+, \\ \textit{pattern fail} & \textit{if } w \in P_-, \\ ? & \textit{otherwise} \end{cases}$$

The suffix matching semantics *of* P *is a function*

$$[\![P]\!]_{suffix} : \mathcal{E}^* \to \{\textit{pattern match}, ?\}$$

defined as follows for each $w \in \mathcal{E}^*$*:*

$$[\![P]\!]_{suffix}(w) = \begin{cases} \textit{pattern match} & \textit{if } \exists w_1, w_2 \textit{ such that } w = w_1 w_2 \textit{ and} \\ & [\![P]\!]_{total}(w_2) = \textit{pattern match} \\ ? & \textit{otherwise} \end{cases}$$

For example, for a regular pattern "A* B", a sequence of events "A B B" will be matched only once at the first "B" event and then cause a failure at the second "B" using the total matching semantics. Using the suffix matching s, the pattern will be matched twice, once for each "B" event: the first matches either the whole trace "A B" or the partial trace consisting of just the first "B" with zero occurrences of "A", while the second matches the subsequent partial trace "B" (the second "B" in the trace) with zero occurrences of "A".

It is relatively easy to support suffix trace matching in a total matching and vice versa. For example, to capture suffix matching in a total matching setting, all one needs to do is to maintain a set of states, while a new monitor state is produced at each event; the set will contain at most as many states as the property monitor can have. Conversely, to capture total trace matching in a suffix matching setting such as Tracematches', all one needs to do is to generate an artificial event only once at the beginning of the trace, say "start", and then automatically change any pattern "P" to "start P". More details about implementing suffix matching in a total matching setting is discussed in Section 5.2.

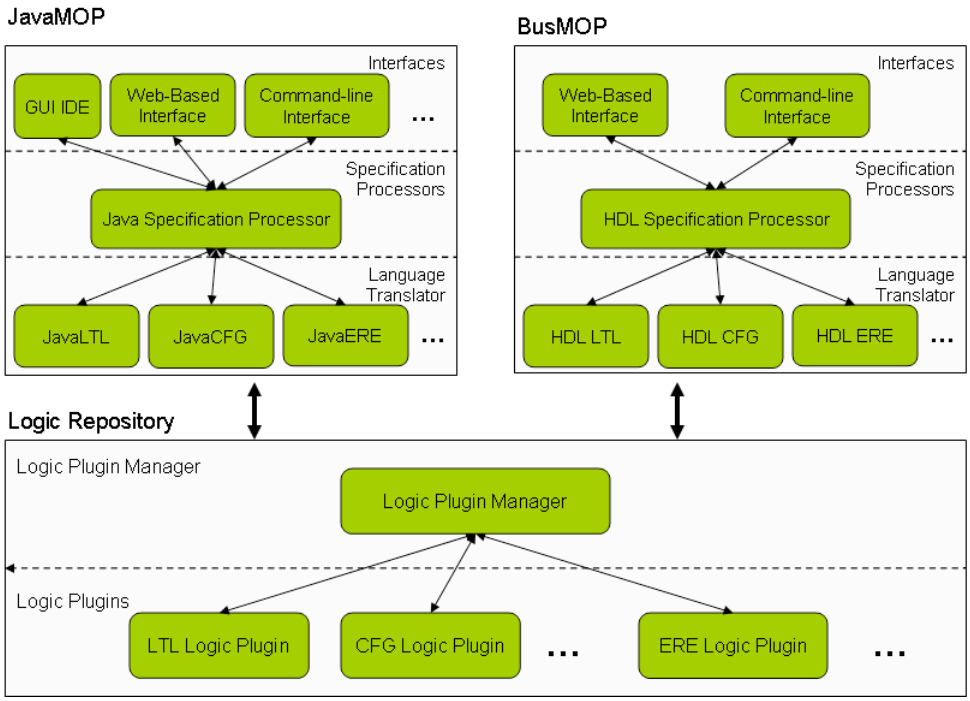

Figure 3.2: MOP architecture

3.3 Generic, Extensible Framework

Figure 3.2 shows the architecture of MOP. The architecture separates monitor generation and integration following the separation of observation and verification in the MOP monitoring model and provides a generic, extensible framework for runtime monitoring. More precisely, there are two kinds of components in MOP, namely logic repositories and language clients. The logic repository, shown in the bottom of Figure 3.2, contains various logic plugins and a logic plugin manager component. The former is the core component to generate monitoring code from formulas written in a specific logic; for example, the PTLTL plugin synthesizes state machines from PTLTL formulas. The output of logic plugins is usually pseudocode and not bound to any specific programming language. This way, the essential monitoring generation can be shared by different instances of MOP using different programming languages. The logic plugin manager bridges the communication between the languages client and the logic plugin. More specifically, it receives the monitor generation request from the language client and distributes the request to an appropriate plugin. After the plugin synthesizes the monitor for the request, the logic plugin manager collects the result and sends it back to the language client. This way, one can easily add new logic plugins into the repository to support new specification formalisms in MOP without changing the language client.

The language client hides the programming-language-independent logic repository and provides language specific support for applying MOP in particular programming languages. Every language client is usually composed of three layers: the bottom layer contains language translators that translate the abstract output of logic plugins into concrete code in a specific programming language; the middle layer is the specification processor that extracts formulas from the given property specification and then instruments the generated monitoring code into the target program; at last, the top layer provides usage interfaces to the user. Presently, two language clients, namely, JavaMOP that is an MOP instance for Java and BusMOP that is an MOP instance for hardware monitoring. This thesis presents only JavaMOP (Chapter 5); more details about BusMOP can be found in [73].

This architecture provides a generic, extensible framework for MOP, in terms of programming languages and specification formalisms. One to instantiate MOP with specific programming languages and specification formalisms to support different domains. In fact, as discussed in Section 8.1, many existing RV systems can be captured as special instances of this framework.

Logic Plugins

Every logic plugin implements and encapsulates a monitor synthesis algorithm for a particular requirements specification formalism. The logic plugin accepts as the input a set of abstract events and a formula written in the underlying formalism and outputs an abstract monitor, usually a piece of pseudocode, which checks a trace of events against the given formula. Presently, we have implemented the following logic plugins in the MOP logic repository (interested readers may refer to corresponding citations for more details about the monitor synthesis algorithm):

- FSM: a simple logic plugin that accepts a finite state machine as the input and output an identical state machine with auxillary information.
- PTLTL [27]: a logic plugin accepts as the input a past-time linear temporal logic formula that can refer to the past of the execution. It outputs a vector of bits representing the state of the monitor, together with a set of equations that update the vector at every received event.
- FTLTL [78]: a logic plugin accepts as the input a future-time linear temporal logic formula that can refer to the future of the execution and outputs an automaton that checks the input formula against an execution trace.
- ERE [77]: a logic plugin accepts extended regular expressions, i.e., ordinary regular expressions extended with negation, and generates automata checking the input regular expressions.

- PTCaRet [79]: a logic plugin accepts past-time linear temporal logic formualae that are extended with calls and returns. Such formulae can be used to specify properties referring to the call stack during the execution. The logic plugin outputs a vector of bits and a stack of vectors as the state of the monitor and a set of equations and stack operations that update the monitor state at every event.

- CFG [67]: a logic plugin accepts context-free grammars and outputs monitoring code based on Action and Goto tables.

Chapter 4

Parametric Monitoring in MOP

More recently, monitoring of parametric specifications, i.e., specifications with free variables, has received increasing interest due to its effectiveness at capturing system behaviors, as shown in the following example about interaction between the classes Map, Collection and Iterator in Java.

Motivating Example: UnsafeMapIterator

Map and Collection implement data structures for mappings and collections, respectively. Iterator is an interface used to enumerate elements in a collection-typed object. One can also enumerate elements in a Map object using Iterator. But, since a Map object contains key-value pairs, one needs to first obtain a collection object that represents the contents of the map, e.g., the set of keys or the set of values stored in the map, and then create an iterator from the obtained collection. An intricate safety property in this usage, according to the Java API specification, is that when the iterator is used to enumerate elements in the map, the contents of the map should not be changed, or unexpected behaviors may occur. A *violating* behavior with regards to this property, which we call UnsafeMapIterator, can be naturally specified using future time linear temporal logic (FTLTL) with parameters: given that m, c, i are objects of Map, Collection and Iterator, respectively, $\forall m, c, i. \diamond (\mathsf{create\_coll}\langle m, c\rangle \wedge \diamond(\mathsf{create\_iter}\langle c, i\rangle \wedge \diamond(\mathsf{update\_map}\langle m\rangle \wedge \diamond \mathsf{use\_iter}\langle i\rangle)))$, where create_coll is creating a collection from a map, create_iter is creating an iterator from a collection, update_map is updating the map, and use_iter is using the iterator; $\diamond$ means eventually in the future. The formula describes the following sequence of actions: Collection c is obtained from a Map m, an iterator i is created from c, m is changed, and then i is accessed. It can also be specified as an ERE pattern: $\forall m, c, i.\mathsf{create\_collupdate\_map}^*\mathsf{create\_iteruse\_iter}^*\mathsf{update\_mapupdate\_map}^*\mathsf{use\_iter}$. When an observed execution satisfies the FTLTL formula or the ERE pattern, the UnsafeMapIterator property is broken in the execution.

It is highly non-trivial to monitor such parametric specifications efficiently. We may see a tremendous number of parameter instances during the execution; for example, it is not uncommon to see

hundreds of thousands of iterators in one execution. Also, some events may contain partial information about parameters, making it more difficult in locating other relevant parameter bindings during the monitoring process; for example, in the above specification, when a $\mathsf{update\_map}\langle m\rangle$ is received, we need to find all $\mathsf{create\_coll}\langle m, c\rangle$ events with the same binding for m, and transitively, all $\mathsf{create\_iter}\langle c, i\rangle$ with the same c as that $\mathsf{create\_coll}$.

Several approaches were introduced to support the monitoring of parametric specifications, including Eagle [15], Tracematches [3, 13], PQL [66] and PTQL [44]. However, they are all limited in terms of supported specification formalisms or viable execution traces. Most techniques, e.g., Eagle, Tracematches, PQL and PTQL, follow a formalism-dependent approach, that is, they have their parametric specification formalisms hardwired, e.g., regular patterns (like Tracematches), context-free patterns (like PQL) with parameters, etc., and then develop algorithms to generate monitoring code for the particular formalisms. Although this approach provides a feasible solution to monitoring parametric specifications, we argue that it not only has limited expressiveness, but also causes unnecessary complexity in developing optimal monitor generation algorithms, often leading to inefficient monitoring. In fact, experiments in Section 5.3 show that our formalism-independent solution generates more efficient monitoring code than other existing tools.

Following the genericity of the MOP framework, we have developed a general technique to build optimized parametric monitors from non-parametric monitors, which is based on a general solution for handling parametric trace. In this novel technique, we apply knowledge about the monitored *property* to improve efficiency. The needed knowledge, encoded as *enable sets*, depends only on the property and not on the formalism in which it is specified. It can be easily computed as a *side effect* when generating a monitor from the property, as discussed in Section 4.4.3. Our experiments show that this technique of optimization based on enable sets, combined with the general parametric trace slicing algorithm, represents the first *efficient, modular* technique for monitoring fully general properties (i.e., the properties do not need to instantiate all the parameters in the creation events or use a fixed logical formalism). In fact, it is *more* efficient than the systems that do use a fixed formalism (see Sectionsec:javamop-eval).

In the rest of this chapter is organized as follows. Section 4.1 gives a high-level overview of our approach. Section 4.2 introduces a general solution for parametric trace slicing that provides the foundation for parametric monitoring. Section 4.3 discusses general algorithsm for parametric monitoring and Section 4.4 proposes enable-set-based optimization for parametric monitoring. The presented technique has been implemented and evaluated in JavaMOP, as discussed in Section 5.2.

4.1 Approach Overview

#	Event	#	Event
1	create_coll$\langle m_1, c_1 \rangle$	7	update_map$\langle m_1 \rangle$
2	create_coll$\langle m_1, c_2 \rangle$	8	use_iter$\langle i_2 \rangle$
3	create_iter$\langle c_1, i_1 \rangle$	9	create_coll$\langle m_2, c_3 \rangle$
4	create_iter$\langle c_1, i_2 \rangle$	10	create_iter$\langle c_3, i_4 \rangle$
5	use_iter$\langle i_1 \rangle$	11	use_iter$\langle i_4 \rangle$
6	create_iter$\langle c_2, i_3 \rangle$		

Figure 4.1: Possible Execution Trace Over the Events Specified in UnsafeMapIterator.

Our approach to monitoring parametric traces against parametric properties is based on the observation that each parametric trace actually contains multiple *non-parametric trace slices*, each for a particular parameter binding instance. The formal definition of the trace slice can be found in Section 4.2.1, but intuitively, a slice of a parametric trace for a particular parameter binding consists of names of all the events that have *less informative* parameter bindings. Informally, a parameter binding b_1 is less informative than a parameter binding b_2 if and only if the parameters for which they have bindings agree, and b_2 binds either an equal number of parameters or more parameters: parameter $\langle m_1, c_2 \rangle$ is less informative than $\langle m_1, c_2, i_3 \rangle$ because the parameters they both bind, m and c, agree on their values, m_1 and c_2, respectively, and $\langle m_1, c_2, i_3 \rangle$ binds one more parameter. Figure 4.2 shows the trace slices and their corresponding parameter bindings contained in the trace in Figure 4.1. The Status column denotes the output category that the slice falls into (for ERE). In this case everything but the slice for $\langle m_1, c_1, i_2 \rangle$, which matches the property, is in the "?" (undecided) category. For example, the trace for the binding $\langle m_1, c_1 \rangle$ contains create_coll update_map (the first and seventh events in the trace) and the trace for the binding $\langle m_1, c_1, i_2 \rangle$ is create_coll create_iter update_map use_iter (the first, fourth, seventh, and eighth events in the trace).

Based on this observation, our approach creates a set of monitor instances during the monitoring

Instance	Slice	Status
$\langle m_1 \rangle$	update_map	?
$\langle m_1, c_1 \rangle$	create_coll update_map	?
$\langle m_1, c_2 \rangle$	create_coll update_map	?
$\langle m_2, c_3 \rangle$	create_coll	?
$\langle m_1, c_1, i_1 \rangle$	create_coll create_iter use_iter update_map	?
$\langle m_1, c_1, i_2 \rangle$	create_coll create_iter update_map use_iter	match
$\langle m_1, c_2, i_3 \rangle$	create_coll create_iter update_map	?
$\langle m_2, c_3, i_4 \rangle$	create_coll create_iter use_iter	?

Figure 4.2: Slices for the Trace in Figure 4.1.

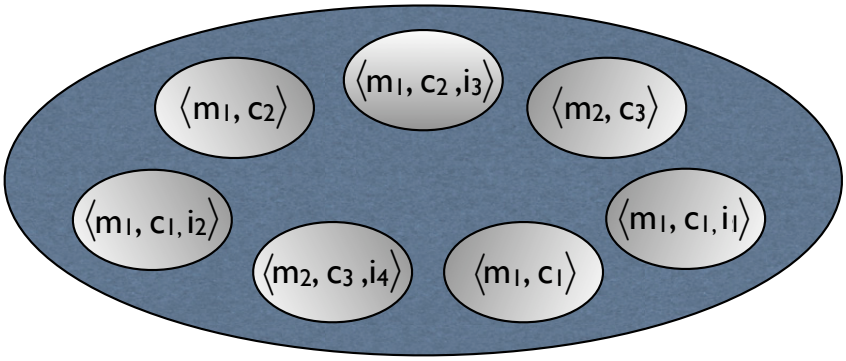

Figure 4.3: A Monitor Set (Parametric Monitor) with Corresponding Parameter Instance Monitors.

process, each handling a trace slice for a parameter binding. Figure 4.3 shows the set of monitors created for the trace in Figure 4.1, each monitor labeled by the corresponding parameter binding. This way, the monitor *does not need to handle the parameter information* and can employ any existing technique for ordinary, non-parametric traces, including state machines and push-down automata, providing a formalism-independent way to check parametric properties. When an event comes, our algorithm will dispatch it to related monitors, which will update their states accordingly. For example, the seventh event in Figure 4.1, update_map$\langle m_1 \rangle$, will be dispatched to monitors for $\langle m_1, c_1 \rangle$, $\langle m_1, c_2 \rangle$, $\langle m_1, c_1, i_1 \rangle$, $\langle m_1, c_1, i_2 \rangle$, and $\langle m_1, c_2, i_3 \rangle$. New monitor instances will be created if the event contains new parameter instances. For example, when the third event in Figure 4.1, create_iter$\langle c_1, i_1 \rangle$, is received, a new monitor will be created for $\langle m_1, c_1, i_1 \rangle$ by combining $\langle m_1, c_1 \rangle$ in the first event with $\langle c_1, i_1 \rangle$. Detailed discussion about the monitoring algorithm can be found in Section 4.4.1.

An algorithm to build parameter instances from observed events, like the one introduced in [31], may create many useless monitor instances leading to prohibitive runtime overheads. For example, Figure 4.2 does not need to contain the binding $\langle m_1, c_3, i_4 \rangle$ even though it can be created by combining the parameter instances of update_map$\langle m_1 \rangle$ (the seventh event) and create_iter$\langle c_3, i_4 \rangle$ (the tenth event). It is safe to ignore this binding here because m_1 is not the underlying map for c_3, i_4. It is critical to minimize the number of monitor instances created during monitoring. The advantage is twofold: (1) that it reduces the needed memory space, and (2), more importantly, monitoring efficiency is improved since fewer monitors are triggered for each received event.

We present an effective solution in this paper to minimize the created monitors, based on the concept of the *enable set*, which is formally discussed in Section 4.4.2. An enable set is constructed for each event, say e, defined for a particular property. The enable set associated with e is a set

of sets of parameters. Each of these sets of parameters denotes parameters that must have been seen before the arrival of event e, for e to be acceptable by a monitor instance. Consider the event update_map, it may occur anywhere in a matching trace, *except* for as the first event. Because the first event must be create_coll in a matching trace, and because create_coll instantiates both m and c, one of the sets in the enable set for update_map must be $\{m, c\}$. However, update_map may (in fact, must, to match the pattern) occur after the create_iter event. Because create_iter many not occur before create_coll we also have the set $\{m, c, i\}$ in the enable set for update_map. The final result for the enable set for update_map is thus: $\{\{m, c\}, \{m, c, i\}\}$. Therefore, when update_map$\langle m_1 \rangle$ arrives (the seventh event), the instance monitors for $\langle m_1, c_1 \rangle$ and $\langle m_1, c_2 \rangle$ must be updated because they bind $\{m, c\}$, and the instance monitors for $\langle m_1, c_1, i_1 \rangle$, $\langle m_1, c_1, i_2 \rangle$, and $\langle m_1, c_2, i_3 \rangle$ must be updated because they bind $\{m, c, i\}$, and have the same value for m (m_1). In this example all of the instances to update have already been created by the time the event arrives, but it should also be noted that no new instances can be created because at least m and c must be bound before update_map can occur.

It is worth mentioning that one may reduce the needed monitors using static program analysis, e.g., the one introduced in [21]. However, such techniques are based on the program targeted for monitoring and lead to drawbacks in practice: (1) it is a more complex and thus slower analysis and (2) the analysis must be run for every target program, making the approach non-modular. For example, if the property to monitor is related to some library, one will have to run the analysis for every program using the library, which can be expensive, and often infeasible. The analysis needed by our approach, on the other hand, is usually much quicker[1], because properties tend to be much smaller than the programs they are designed to monitor. Moreover, our optimization technique requires no additional analysis when used in a situation, like for a library, where a property is checked for different programs, because the enable set is derived from the property itself instead of the targeted program.

4.2 Background: Parametric Trace Slicing

We next formalize the notions of parametric traces and properties and then introduce a general algorithm to slice a parametric trace into a set of non-parametric trace slices.

[1]The analysis is upper bounded by the number of acyclic paths from the start state/symbol through a finite state machine/context free grammar, because convergence is achieved through one cycle. Finite state machines and context free grammars for properties tend to be small.

4.2.1 Parametric Traces and Properties

Here we introduce the notions of event, trace and property, first non-parametric and then parametric. Traces are sequences of events. Parametric events can carry data-values, as instances of parameters. Parametric traces are traces over parametric events. Properties are trace classifiers, that is, mappings partitioning the space of traces into categories (violating traces, validating traces, don't know traces etc.). Parametric properties are parametric trace classifiers and provide, for each parameter instance, the category to which the trace slice corresponding to that parameter instance belongs. Trace slicing is defined as a reduct operation that forgets all the events that are unrelated to the given parameter instance.

The Non-Parametric Case

Definition 4 *Let $\mathcal{E}$ be a set of (non-parametric) events, called **base events** or simply **events**. An $\mathcal{E}$-**trace**, or simply a (non-parametric) **trace** when $\mathcal{E}$ is understood or not important, is any finite sequence of events in $\mathcal{E}$, that is, an element in $\mathcal{E}^*$. If event $e \in \mathcal{E}$ appears in trace $w \in \mathcal{E}^*$ then we write $e \in w$.*

Our parametric trace slicing and monitoring techniques in Sections 4.2.4 and 4.3.2 can be easily adapted to also work with infinite traces. Since infinite versus finite traces is not an important aspect of the work reported here, we keep the presentation free of unnecessary technical complications and consider only finite traces.

Example. *(part 1 of simple running example)* Consider a certain resource (e.g., a synchronization object) that can be acquired and released during the lifetime of a given procedure (between its begin and end). Then $\mathcal{E} = \{\mathsf{acquire}, \mathsf{release}, \mathsf{begin}, \mathsf{end}\}$ and execution traces corresponding to this resource are sequences of the form "begin acquire acquire release end begin end", "begin acquire acquire", "begin acquire release acquire end", etc. For now there are no "good" or "bad" execution traces. □

There is a plethora of formalisms to specify trace requirements. Many of these result in specifying at least two types of traces: those *validating* the specification (i.e, correct traces), and those *violating* the specification (i.e., incorrect traces).

Example. *(part 2)* Consider a regular expression specification, $(\mathsf{begin}(\epsilon \mid (\mathsf{acquire}(\mathsf{acquire} \mid \mathsf{release})^* \mathsf{release}))\mathsf{end})^*$, stating that the procedure can (non-recursively) take place multiple times and, if the resource is acquired during the procedure then it is released by the end of the procedure. Assume that the resource can be acquired and released multiple times, with the effect of acquiring and

respectively releasing it precisely once; regular expressions cannot specify matched acquire/release events, we are going to do so using context-free patterns in the next section. The validating traces for this property are those satisfying the pattern, e.g., "begin acquire acquire release end begin end". At first sight, one may say that all the other traces are violating traces, because they are not in the language of the regular expression. However, there are two interesting types of such "violating" traces: ones which may still lead to a validating trace provided the right events will be received in the future, e.g., "begin acquire acquire", and ones which have no chance of becoming a validating trace, e.g. "begin acquire release acquire end". □

In general, traces are not enforced to correspond to terminated programs (this is particularly useful in monitoring); if one wants to enforce traces to correspond to terminated programs, then one can have the system generate a special "end-of-trace" event and have the property specification require that event at the end of each trace.

Therefore, a trace property may partition the space of traces into more than two categories. For some specification formalisms, for example ones based on fuzzy logics or multiple truth values, the set of traces may be split into more than three categories, even into a continuous space of categories.

Definition 5 *An $\mathcal{E}$-**property** P, or simply a (base or non-parametric) **property**, is a function $P : \mathcal{E}^* \to \mathcal{C}$ partitioning the set of traces into categories $\mathcal{C}$. It is common, though not enforced, that $\mathcal{C}$ includes "validating", "violating", and "don't know" (or "?") categories. In general, $\mathcal{C}$, the co-domain of P, can be any set.*

We believe that the definition of non-parametric trace property above is general enough that it can easily accommodate any particular specification formalism, such as ones based on linear temporal logics, regular expressions, context-free grammars, etc. All one needs to do in order to instantiate the general results in this paper for a particular specification formalism is to decide upon the desired categories in which traces are intended to be classified, and then define the property associated to a specification accordingly.

For example, if the specification formalism of choice is that of regular expressions over $\mathcal{E}$ and one is interested in classifying traces in three categories as in our example above, then one can pick $\mathcal{C}$ to be the set $\{\mathsf{validating}, \mathsf{violating}, \mathsf{don't\ know}\}$ and, for a given regular expression E, define its associated property $P_E : \mathcal{E}^* \to \mathcal{C}$ as follows: $P_E(w) = \mathsf{validating}$ iff w is in the language of E, $P_E(w) = \mathsf{violating}$ iff there is no $w' \in \mathcal{E}^*$ such that $w\,w'$ is in the language of E, and $P_E(w) = \mathsf{don't\ know}$ otherwise; this is the monitoring semantics of regular expressions in JavaMOP [29].

Other semantic choices are possible even for the simple case of regular expressions; for example, one may choose $\mathcal{C}$ to be the set $\{\mathsf{matching}, \mathsf{don't\ care}\}$ and define $P_E(w) = \mathsf{matching}$ iff w is in the language of E, and $P_E(w) = \mathsf{don't\ care}$ otherwise; this is the semantics of regular expressions in Tracematches [3], where, depending upon how one writes the regular expression, matching can mean either a violation or a validation of the desired property.

In some applications, one may not be interested in certain categories of traces, such as in those classified as $\mathsf{don't\ know}$ or $\mathsf{don't\ care}$; if that is the case, then those applications can simply ignore these, like Tracematches and JavaMOP do. It may be worth making it explicit that in this paper we do not attempt to propose or promote any particular formalism for specifying properties about execution traces. Instead, our approach is to define properties as generally as possible to capture the various specification formalisms that we are aware of as special cases, and then to develop our subsequent techniques to work with such general properties.

An additional benefit of defining properties so generally, as mappings from traces to categories, is that parametric properties, in spite of their much more general flavor, are also properties (but, obviously, over different traces and over different categories).

The Parametric Case

Events often carry concrete data instantiating abstract parameters.

Example. *(part 3)* In our running example, events $\mathsf{acquire}$ and $\mathsf{release}$ are parametric in the resource being acquired or released; if r is the name of the generic "resource" parameter and r_1 and r_2 are two concrete resources, then parametric acquire/release events have the form $\mathsf{acquire}\langle r \mapsto r_1\rangle$, $\mathsf{release}\langle r \mapsto r_2\rangle$, etc. Not all events need carry instances for all parameters; e.g., the begin/end parametric events have the form $\mathsf{begin}\langle\bot\rangle$ and $\mathsf{end}\langle\bot\rangle$, where $\bot$, the partial map undefined everywhere, instantiates no parameter. □

We let $[A \to B]$ and $[A \rightharpoonup B]$ denote the sets of total and respectively partial functions from A to B.

Definition 6 ***(Parametric events and traces).*** *Let X be a set of* ***parameters*** *and let V_X be a set of corresponding* ***parameter values****. If $\mathcal{E}$ is a set of base events like in Definition 4, then let $\mathcal{E}\langle X\rangle$ denote the set of corresponding* ***parametric events*** *$e\langle\theta\rangle$, where e is a base event in $\mathcal{E}$ and θ is a partial function in $[X \rightharpoonup V]$. A* ***parametric trace*** *is a trace with events in $\mathcal{E}\langle X\rangle$, that is, a word in $\mathcal{E}\langle X\rangle^*$.*

Therefore, a parametric event is an event carrying values for zero, one, several or even all the parameters, and a parametric trace is a finite sequence of parametric events. In practice, the number of values carried by an event is finite; however, we do not need to enforce this restriction in our theoretical developments. Also, in practice the parameters may be typed, in which case the set of their corresponding values is given by their type. To simplify writing, we occasionally assume the set of parameter values V_X implicit.

Example. *(part 4)* A parametric trace for our running example can be the following: $\mathsf{begin}\langle\bot\rangle$ $\mathsf{acquire}\langle\theta_1\rangle$ $\mathsf{acquire}\langle\theta_2\rangle$ $\mathsf{acquire}\langle\theta_1\rangle$ $\mathsf{release}\langle\theta_1\rangle$ $\mathsf{end}\langle\bot\rangle$ $\mathsf{begin}\langle\bot\rangle$ $\mathsf{acquire}\langle\theta_2\rangle$ $\mathsf{release}\langle\theta_2\rangle$ $\mathsf{end}\langle\bot\rangle$, where θ_1 maps r to r_1 and θ_2 maps r to r_2. To simplify writing, we only list the parameter instance values when writing parameter instances, that is, $\langle r_1\rangle$ instead of $\langle r \mapsto r_1\rangle$, or $\tau{\restriction_{r_2}}$ instead of $\tau{\restriction_{r\mapsto r_2}}$, etc. With this notation, the above trace becomes: $\mathsf{begin}\langle\rangle$ $\mathsf{acquire}\langle r_1\rangle$ $\mathsf{acquire}\langle r_2\rangle$ $\mathsf{acquire}\langle r_1\rangle$ $\mathsf{release}\langle r_1\rangle$ $\mathsf{end}\langle\rangle$ $\mathsf{begin}\langle\rangle$ $\mathsf{acquire}\langle r_2\rangle$ $\mathsf{release}\langle r_2\rangle$ $\mathsf{end}\langle\rangle$. This trace involves two resources, r_1 and r_2, and it really consists of *two trace slices*, one for each resource, merged together. The begin and end events belong to both trace slices. The slice corresponding to θ_1 is "begin acquire acquire release end begin end", while the one for θ_2 is "begin acquire end begin acquire release end ". □

Definition 7 ***(Trace slicing)*** *Given parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$ and partial function θ in $[X \rightharpoonup V]$, we let the θ-**trace slice** $\tau{\restriction_\theta} \in \mathcal{E}^*$ be the non-parametric trace in $\mathcal{E}^*$ defined as follows:*

- $\epsilon{\restriction_\theta} = \epsilon$, *where ϵ is the empty trace/word, and*
- $(\tau\, e\langle\theta'\rangle){\restriction_\theta} = \begin{cases} (\tau{\restriction_\theta})\, e & \text{when } \theta' \sqsubseteq \theta \\ \tau{\restriction_\theta} & \text{when } \theta' \not\sqsubseteq \theta \end{cases}$,

where $\theta' \sqsubseteq \theta$ iff for any $x \in X$, if $\theta'(x)$ is defined then $\theta(x)$ is also defined and $\theta'(x) = \theta(x)$.

Therefore, the trace slice $\tau{\restriction_\theta}$ first filters out all the parametric events that are not relevant for the instance θ, i.e., which contain instances of parameters that θ does not care about, and then, for the remaining events relevant to θ, it forgets the parameters so that the trace can be checked against base, non-parametric properties.

Specifying properties over parametric traces is rather challenging, because one may want to specify a property for one generic parameter instance and then say "and so on for all the other instances". In other words, one may want to specify a universally quantified property over base events, but, unfortunately, the underlying specification formalism may not allow universal quantification over data; for example, none of the conventional formalisms to specify properties on linear traces listed

above (i.e, linear temporal logics, regular expressions, context-free grammars) or mentioned in the rest of the paper has universal data quantification.

Definition 8 *Let X be a set of parameters together with their corresponding parameter values V_X, like in Definition 6, and let $P : \mathcal{E}^* \to \mathcal{C}$ be a non-parametric property like in Definition 5. Then we define the* ***parametric property*** *$\Lambda X.P$ as the property (over traces $\mathcal{E}\langle X\rangle^*$ and categories $[[X \rightharpoonup V] \to \mathcal{C}]$)*

$$\Lambda X.P : \mathcal{E}\langle X\rangle^* \to [[X \rightharpoonup V] \to \mathcal{C}]$$

defined as $(\Lambda X.P)(\tau)(\theta) = P(\tau\restriction_\theta)$ for any $\tau \in \mathcal{E}\langle X\rangle^*$ and any $\theta \in [X \rightharpoonup V]$. If $X = \{x_1, ..., x_n\}$ we may write $\Lambda x_1, ..., x_n.P$ instead of $(\Lambda\{x_1, ..., x_n\}.P$. Also, if P_φ is defined using a pattern or formula φ in some particular trace specification formalism, we take the liberty to write $\Lambda X.\varphi$ instead of $\Lambda X.P_\varphi$.

Parametric properties $\Lambda X.P$ over base properties $P : \mathcal{E}^* \to \mathcal{C}$ are therefore properties taking traces in $\mathcal{E}\langle X\rangle^*$ to categories $[[X \rightharpoonup V] \to \mathcal{C}]$, i.e., function domains from parameter instances to base property categories. $\Lambda X.P$ is defined as if many instances of P are observed at the same time on the parametric trace, one property instance for each parameter instance, each property instance concerned with its events only, dropping the unrelated ones.

Example. *(part 5)* Let P be the non-parametric property specified by the regular expression in the second part of our running example above (using the mapping of regular expressions to properties discussed in the second part of our running example and after Definition 5 – i.e., the JavaMOP semantic approach to parametric monitoring [29]). Since we want P to hold for any resource instance, we then define the following parametric property (i.e., $\Lambda r.P$):

$$\Lambda r.(\mathsf{begin}\ (\epsilon \mid (\mathsf{acquire}\ (\mathsf{acquire} \mid \mathsf{release})^*\ \mathsf{release}))\ \mathsf{end})^*.$$

If τ is the parametric trace and θ_1 and θ_2 are the parameter instances in the fourth part of our running example, then the semantics of the parametric property above on trace τ is validating for parameter instance θ_1 and violating for parameter instance θ_2. □

4.2.2 Least Upper Bound Closures of Partial Maps

In this section we first discuss some basic notions of partial functions and least upper bounds of them, then we introduce least upper bounds of sets of partial functions and least upper bound closures of

sets of partial functions. This section is rather mathematical. We need these mathematical notions because, as already seen, parameter instances are partial maps from the domain of parameters to the domain of parameter values. As shown later, whenever a new parametric event is observed, it needs to be dispatched to the interested parts (trace slices or monitors), and those parts updated accordingly: these informal operations can be rigorously formalized as existence of least upper bounds and least upper bound closures over parameter instances, i.e., partial functions.

Partial Functions

We think of partial functions as "information carriers": if a partial function θ is defined on an element x of its domain, then "θ carries the information $\theta(x)$ about $x \in X$". Some partial functions can carry more information than others; two or more partial functions can, together, carry compatible information, but can also carry incompatible information (when two or more of them disagree on the information they carry for a particular $x \in X$). Recall that $[X \rightarrow V_X]$ and $[X \rightharpoonup V]$ represent the sets of *total* and of *partial functions* from X to V_X, respectively.

Definition 9 *The domain of $\theta \in [X \rightharpoonup V]$ is the set $Dom(\theta) = \{x \in X \mid \theta(x)$ defined$\}$. Let $\perp \in [X \rightharpoonup V]$ be the map undefined everywhere, that is, $Dom(\perp) = \emptyset$. If $\theta, \theta' \in [X \rightharpoonup V]$ then we say that θ **is less informative than** θ', written $\theta \sqsubseteq \theta'$, if for any $x \in X$, $\theta(x)$ defined implies $\theta'(x)$ also defined and $\theta'(x) = \theta(x)$.*

It is known that $([X \rightharpoonup V], \sqsubseteq, \perp)$ is a complete (i.e., any $\sqsubseteq$-chain has a least upper bound) partial order with bottom (i.e., $\perp$).

Definition 10 *Given $\Theta \subseteq [X \rightharpoonup V]$ and $\theta' \in [X \rightharpoonup V]$,*

- *θ' is an **upper bound** of Θ iff $\theta \sqsubseteq \theta'$ for any $\theta \in \Theta$; Θ **has upper bounds** iff there is a θ' which is an upper bound of Θ;*
- *θ' is the **least upper bound (lub)** of Θ iff θ' is an upper bound of Θ and $\theta' \sqsubseteq \theta''$ for any other upper bound θ'' of Θ;*
- *θ' is the **maximum (max)** of Θ iff $\theta' \in \Theta$ and θ' is a lub of Θ.*

Intuitively, a set of partial functions has an upper bound iff the partial functions in the set are *compatible*, that is, no two of them disagree on the value of a particular element in their domain. Least upper bounds and maximums may not always exist for any $\Theta \subseteq [X \rightharpoonup V]$; if a lub or a maximum for Θ exists, then it is, of course, unique ($\sqsubseteq$ is a partial order, so antisymmetric).

Definition 11 *Given $\Theta \subseteq [X \rightharpoonup V]$, let $\sqcup\Theta$ and $\max\Theta$ be the lub and the max of Θ, respectively, when they exist. When Θ is finite, one may write $\theta_1 \sqcup \theta_2 \sqcup \cdots \sqcup \theta_n$ instead of $\sqcup\{\theta_1, \theta_2, \ldots, \theta_n\}$.*

If Θ has a maximum, then it also has a lub and $\sqcup\Theta = \max\Theta$. Here are several common properties that we use frequently:

Proposition 1 *The following hold ($\theta, \theta_1, \theta_2, \theta_3 \in [X \rightharpoonup V]$): $\bot \sqcup \theta$ exists and $\bot \sqcup \theta = \theta$; $\theta_1 \sqcup \theta_2$ exists iff $\theta_2 \sqcup \theta_1$ exists, and, if they exist then $\theta_1 \sqcup \theta_2 = \theta_2 \sqcup \theta_1$; $\theta_1 \sqcup (\theta_2 \sqcup \theta_3)$ exists iff $(\theta_1 \sqcup \theta_2) \sqcup \theta_3$ exists, and if they exist then $\theta_1 \sqcup (\theta_2 \sqcup \theta_3) = (\theta_1 \sqcup \theta_2) \sqcup \theta_3$.*

Proposition 2 *Let $\Theta \subseteq [X \rightharpoonup V]$. Then*

1. *Θ has an upper bound iff for any $\theta_1, \theta_2 \in \Theta$ and $x \in X$, if $\theta_1(x)$ and $\theta_2(x)$ defined then $\theta_1(x) = \theta_2(x)$;*

2. *If Θ has an upper bound then $\sqcup\Theta$ exists and, for any $x \in X$,*
$$(\sqcup\Theta)(x) = \begin{cases} \textit{undefined} & \textit{if } \theta(x) \textit{ undefined for any } \theta \in \Theta \\ \theta(x) & \textit{if there is a } \theta \in \Theta \textit{ with } \theta(x) \textit{ defined.} \end{cases}$$

Proof: Since Θ has an upper bound $\theta' \in [X \rightharpoonup V]$ iff $\theta \sqsubseteq \theta'$ for any $\theta \in \Theta$, if $\theta_1, \theta_2 \in \Theta$ and $x \in X$ with $\theta_1(x)$ and $\theta_2(x)$ defined then $\theta'(x)$ is also defined and $\theta_1(x) = \theta_2(x) = \theta'(x)$. Suppose now that for any $\theta_1, \theta_2 \in \Theta$ and $x \in X$, if $\theta_1(x)$ and $\theta_2(x)$ defined then $\theta_1(x) = \theta_2(x)$. All we need to show in order to prove both results is that we can find a lub for Θ. Let $\theta' \in [X \rightharpoonup V]$ be defined as follows: for any $x \in X$, let
$$\theta'(x) = \begin{cases} \text{undefined} & \text{if } \theta(x) \text{ undefined for any } \theta \in \Theta \\ \theta(x) & \text{if there is a } \theta \in \Theta \text{ such that } \theta(x) \text{ defined} \end{cases}$$
First, note that θ' above is indeed well-defined, because we assumed that for any $\theta_1, \theta_2 \in \Theta$ and $x \in X$, if $\theta_1(x)$ and $\theta_2(x)$ defined then $\theta_1(x) = \theta_2(x)$. Second, note that θ' is an upper bound for Θ: indeed, if $\theta \in \Theta$ and $x \in X$ such that $\theta(x)$ defined, then $\theta'(x)$ is also defined and $\theta'(x) = \theta(x)$, that is, $\theta \sqsubseteq \theta'$ for any $\theta \in \Theta$. Finally, θ' is a lub for Θ: if θ'' is another upper bound for Θ and $\theta'(x)$ defined for some $x \in X$, that is, $\theta(x)$ defined for some $\theta \in \Theta$ and $\theta'(x) = \theta(x)$, then $\theta''(x)$ also defined and $\theta'(x) = \theta(x)$ (as $\theta \sqsubseteq \theta''$), so $\theta' \sqsubseteq \theta''$. □

Proposition 3 *The following hold:*

1. *The empty set of partial functions $\emptyset \subseteq [X \rightharpoonup V]$ has upper bounds and $\sqcup\emptyset = \bot$;*

2. *The one-element sets have upper bounds and $\sqcup\{\theta\} = \theta$ for any $\theta \in [X \rightharpoonup V]$;*

3. *The bottom "$\perp$" does not influence the least upper bounds:* $\sqcup(\{\perp\} \cup \Theta) = \sqcup\Theta$ *for any* $\Theta \subseteq [X \rightharpoonup V]$*;*

4. *If* $\Theta, \Theta' \subseteq [X \rightharpoonup V]$ *s.t.* $\sqcup\Theta'$ *exists and for any* $\theta \in \Theta$ *there is a* $\theta' \in \Theta'$ *with* $\theta \sqsubseteq \theta'$*, then* $\sqcup\Theta$ *exists and* $\sqcup\Theta \sqsubseteq \sqcup\Theta'$*; e.g., if* $\sqcup\Theta'$ *exists and* $\Theta \subseteq \Theta'$ *then* $\sqcup\Theta$ *exists and* $\sqcup\Theta \sqsubseteq \sqcup\Theta'$*;*

5. *Let* $\{\Theta_i\}_{i \in I}$ *be a family of sets of partial functions with* $\Theta_i \subseteq [X \rightharpoonup V]$*. Then* $\sqcup \cup \{\Theta_i \mid i \in I\}$ *exists iff* $\sqcup\{\sqcup\Theta_i \mid i \in I\}$ *exists, and, if both exist,* $\sqcup \cup \{\Theta_i \mid i \in I\} = \sqcup\{\sqcup\Theta_i \mid i \in I\}$.

Proof: *1.*, *2.* and *3.* are straightforward. For *4.*, since for each $\theta \in \Theta$ there is some $\theta' \in \Theta'$ with $\theta \sqsubseteq \theta'$, and since $\theta' \sqsubseteq \sqcup\Theta'$ for any $\theta' \in \Theta'$, it follows that $\theta \sqsubseteq \sqcup\Theta'$ for any $\theta \in \Theta$, that is, that $\sqcup\Theta'$ is an upper bound for Θ. Therefore, by Proposition 2 it follows that $\sqcup\Theta$ exists and $\sqcup\Theta \sqsubseteq \sqcup\Theta'$ (the latter because $\sqcup\Theta$ is the *least* upper bound of Θ). We prove *5.* by double implication, each implication stating that if one of the lub's exist then the other one also exists and one of the inclusions holds; that indeed implies that one of the lub's exists if and only if the other one exists and, if both exist, then they are equal. Suppose first that $\sqcup \cup \{\Theta_i \mid i \in I\}$ exists, that is, that $\cup\{\Theta_i \mid i \in I\}$ has an upper bound, say u. Since $\Theta_i \subseteq \cup\{\Theta_i \mid i \in I\}$ for each $i \in I$, it follows first that each Θ_i also has u as an upper bound, so all $\sqcup\Theta_i$ for all $i \in I$ exist, and second by *4.* above that $\sqcup\Theta_i \sqsubseteq \sqcup \cup \{\Theta_i \mid i \in I\}$ for each $i \in I$. Item *4.* above then further implies that $\sqcup\{\sqcup\Theta_i \mid i \in I\}$ exists and $\sqcup\{\sqcup\Theta_i \mid i \in I\} \sqsubseteq \sqcup\{\sqcup \cup \{\Theta_i \mid i \in I\}\} = \sqcup \cup \{\Theta_i \mid i \in I\}$ (the last equality follows by *2.* above). Conversely, suppose now that $\sqcup\{\sqcup\Theta_i \mid i \in I\}$ exists. Since for each $\theta \in \cup\{\Theta_i \mid i \in I\}$ there is some $i \in I$ such that $\theta \sqsubseteq \sqcup\Theta_i$ (an $i \in I$ such that $\theta \in \Theta_i$), item *4.* above implies that $\sqcup \cup \{\Theta_i \mid i \in I\}$ also exists and $\sqcup \cup \{\Theta_i \mid i \in I\} \sqsubseteq \sqcup\{\sqcup\Theta_i \mid i \in I\}$. □

Least Upper Bounds of Families of Sets of Partial Maps

The notions of partial function, upper bound and least upper bound above are broadly known, and many of their properties are folklore. Motivated by requirements and optimizations of our trace slicing and monitoring algorithms in Sections 4.2.4 and 4.3.2, we next define several less known notions. We are actually not aware of other places where these notions are defined, so they could be novel.

We first extend the notion of lub from one associating a partial function to a set of partial functions to one associating a set of partial functions to a family (or set) of sets of partial functions:

Definition 12 *If* $\{\Theta_i\}_{i \in I}$ *is a family of sets in* $[X \rightharpoonup V]$*, then we let the* ***least upper bound*** *(also* ***lub****) of* $\{\Theta_i\}_{i \in I}$ *be defined as:*

$\sqcup\{\Theta_i \mid i \in I\} \stackrel{\text{def}}{=} \{\sqcup\{\theta_i \mid i \in I\} \mid \theta_i \in \Theta_i \textit{ for each } i \in I$

$$\textit{s.t. } \sqcup\{\theta_i \mid i \in I\} \textit{ exists}\}.$$

As before, we use the infix notation when I is finite, e.g., we may write $\Theta_1 \sqcup \Theta_2 \sqcup \ldots \sqcup \Theta_n$ instead of $\sqcup\{\Theta_i \mid i \in \{1, 2, \ldots, n\}\}$.

Therefore, $\sqcup\{\Theta_i \mid i \in I\}$ is the set containing all the lub's corresponding to sets formed by picking for each $i \in I$ precisely one element from Θ_i. Unlike for sets of partial functions, the lub's of families of sets of partial functions always exist; $\sqcup\{\Theta_i \mid i \in I\}$ is the empty set when no collection of $\theta_i \in \Theta_i$ can be found (one $\theta_i \in \Theta_i$ for each $i \in I$) such that $\{\theta_i \mid i \in I\}$ has an upper bound.

There is an admitted slight notational ambiguity between the two least upper bound notations introduced so far. We prefer to purposely allow this ambiguity instead of inventing a new notation for the lub's of families of sets, hoping that the reader is able to quickly disambiguate the two by checking the types of the objects involved in the lub: if partial functions then the first lub is meant, if sets of partial functions then the second. Note that such notational ambiguities are actually common practice elsewhere; e.g., in a monoid $(M, \_*\_ : M \times M \to M, 1)$ with binary operation $*$ and unit 1, the $*$ is commonly extended to sets of elements M_1, M_2 in M as expected: $M_1 * M_2 = \{m_1 * m_2 \mid m_1 \in M_1, m_2 \in M_2\}$.

Proposition 4 *The next hold ($\Theta, \Theta_1, \Theta_2, \Theta_3 \subseteq [X \rightharpoonup V]$):*

1. $\sqcup\emptyset = \{\bot\}$, *where, in this case,* $\emptyset \subseteq \mathcal{P}([X \rightharpoonup V])$;
2. $\sqcup\{\Theta\} = \Theta$; *in particular* $\sqcup\{\emptyset\} = \emptyset$, *where* $\emptyset \subseteq [X \rightharpoonup V]$;
3. $\sqcup\{\{\theta\} \mid \theta \in \Theta\} = \begin{cases} \{\sqcup\Theta\} & \textit{if } \Theta \textit{ has a lub, and} \\ \emptyset & \textit{if } \Theta \textit{ does not have a lub;} \end{cases}$
4. $\emptyset \sqcup \Theta = \emptyset$, *where* $\emptyset \subseteq [X \rightharpoonup V]$;
5. $\{\bot\} \sqcup \Theta = \Theta$;
6. *If* $\Theta_1 \subseteq \Theta_2$ *then* $\Theta_1 \sqcup \Theta_3 \subseteq \Theta_2 \sqcup \Theta_3$; *in particular, if* $\bot \in \Theta_2$ *then* $\Theta_3 \subseteq \Theta_2 \sqcup \Theta_3$;
7. $(\Theta_1 \cup \Theta_2) \sqcup \Theta_3 = (\Theta_1 \sqcup \Theta_3) \cup (\Theta_2 \sqcup \Theta_3)$.

Proof: Recall that the least upper bound $\sqcup\{\Theta_i \mid i \in I\}$ of sets of sets of partial functions is built by collecting all the least upper bounds of sets $\{\theta_i \mid i \in I\}$ containing one element θ_i from each of the involved sets Θ_i. When $|I| = 0$, that is when I is empty, there is precisely one set $\{\theta_i \mid i \in I\}$, the empty set of partial functions. Then *1.* follows by *1.* in Proposition 3. When $|I| = 1$, that is

when $\{\Theta_i \mid i \in I\} = \{\Theta\}$ for some $\Theta \subseteq [X \rightharpoonup V]$ like in *2.*, then the sets $\{\theta_i \mid i \in I\}$ are precisely the singleton sets corresponding to the elements of Θ, so *2.* follows by *2.* in Proposition 3. *3.* holds because there is only one way to pick an element from each singleton set $\{\theta\}$, namely to pick the θ itself; this also shows how the notion of a lub of a family of sets generalizes the conventional notion of lub. When $|I| \geq 2$ and at least one of the involved sets of partial functions is empty, like in *4.*, then there is no set $\{\theta_i \mid i \in I\}$, so the least upper bound of the set of sets is empty (regarded, again, as the empty set of sets of partial functions). *5.* follows by *1.* in Proposition 1. The first part of *6.* is immediate and the second part follows from the first using *5.*. Finally, *7.* follows by double implication: $(\Theta_1 \sqcup \Theta_3) \cup (\Theta_2 \sqcup \Theta_3) \subseteq (\Theta_1 \cup \Theta_2) \sqcup \Theta_3$ follows by *6.* because Θ_1 and Θ_2 are included in $\Theta_1 \cup \Theta_2$, and $(\Theta_1 \cup \Theta_2) \sqcup \Theta_3 \subseteq (\Theta_1 \sqcup \Theta_3) \cup (\Theta_2 \sqcup \Theta_3)$ because for any $\theta_1 \in \Theta_1 \cup \Theta_2$, say $\theta_1 \in \Theta_1$, and any $\theta_3 \in \Theta_3$, if $\theta_1 \sqcup \theta_3$ exists then it also belongs to $\Theta_1 \sqcup \Theta_3$. □

Proposition 5 *Let $\{\Theta_i\}_{i \in I}$ be a family of sets of partial maps in $[X \rightharpoonup V]$ and let $\mathcal{I} = \{I_j\}_{j \in J}$ be a partition of I: $I = \cup\{I_j \mid j \in J\}$ and $I_{j_1} \cap I_{j_2} = \emptyset$ for any different $j_1, j_2 \in J$. Then*

$$\sqcup\{\Theta_i \mid i \in I\} = \sqcup\{\sqcup\{\Theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}.$$

Proof: For each $j \in J$, let Q_j denote the set $\sqcup\{\Theta_{i_j} \mid i_j \in I_j\}$. Definition 12 then implies the following: $Q_j \stackrel{\text{def}}{=} \{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid \theta_{i_j} \in \Theta_{i_j} \text{ for each } i_j \in I_j, \text{ such that } \sqcup\{\theta_{i_j} \mid i_j \in I_j\} \text{ exists}\}$. Definition 12 also implies the following: $\sqcup\{Q_j \mid j \in J\} \stackrel{\text{def}}{=} \{\sqcup\{q_j \mid j \in J\} \mid q_j \in Q_j \text{ for each } j \in J, \text{ such that } \sqcup\{q_j \mid j \in J\} \text{ exists}\}$. Putting the two equalities above together, we get that $\sqcup\{\sqcup\{\Theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$ equals the following:

$$\begin{array}{l} \{\ \sqcup\{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\} \\ \qquad \mid \theta_{i_j} \in \Theta_{i_j} \text{ for each } j \in J \text{ and } i_j \in I_j, \text{ such that} \\ \qquad \sqcup\{\theta_{i_j} \mid i_j \in I_j\} \text{ exists for each } j \in J \text{ and} \\ \qquad \sqcup\{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\} \text{ exists}\}. \end{array}$$

Since $\{I_j\}_{j \in J}$ is a partition of I, the indices i_j generated by "for each $j \in J$ and $i_j \in I_j$" cover precisely all the indices $i \in I$. Moreover, picking partial functions $\theta_{i_j} \in \Theta_{i_j}$ for each $j \in J$ and $i_j \in I_j$ is equivalent to picking partial functions $\theta_i \in \Theta_i$ for each $i \in I$, and, in this case, $\{\theta_i \mid i \in I\} = \cup\{\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$. By *5.* in Proposition 3 we then infer that $\sqcup\{\theta_i \mid i \in I\}$ exists if and only if $\sqcup\{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$ exists, and if both exist then $\sqcup\{\theta_i \mid i \in I\} = \sqcup\{\sqcup\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$; if both exist then $\sqcup\{\theta_{i_j} \mid i_j \in I_j\}$ also exists for each $j \in J$ (because $\{\theta_{i_j} \mid i_j \in I_j\} \subseteq \{\theta_i \mid i \in$

$I\} = \cup\{\{\theta_{i_j} \mid i_j \in I_j\} \mid j \in J\})$. Therefore, we can conclude that $\sqcup\{\sqcup\{\Theta_{i_j} \mid i_j \in I_j\} \mid j \in J\}$ equals $\{\sqcup\{\theta_i \mid i \in I\} \mid \theta_i \in \Theta_i$ for each $i \in I$, such that $\sqcup\{\theta_i \mid i \in I\}$ exists$\}$, which is nothing but $\sqcup\{\Theta_i \mid i \in I\}$. □

Corollary 1 *The following hold:*

1. $\{\bot\} \sqcup \Theta = \Theta$ *(already proved as 5. in Proposition 4);*
2. $\Theta_1 \sqcup \Theta_2 = \Theta_2 \sqcup \Theta_1$*;*
3. $\Theta_1 \sqcup (\Theta_2 \sqcup \Theta_3) = (\Theta_1 \sqcup \Theta_2) \sqcup \Theta_3$*;*

Proof: These follow from Proposition 5 for various index sets I and partitions of it: for *1.* take $I = \{1\}$ and its partition $I = \emptyset \cup I$, take $\Theta_1 = \Theta$, and then use *1.* in Proposition 4 saying that $\sqcup\emptyset = \{\bot\}$; for *2.* take partitions $\{1\} \cup \{2\}$ and $\{2\} \cup \{1\}$ of $I = \{1, 2\}$, getting $\Theta_1 \sqcup \Theta_2 = \Theta_2 \sqcup \Theta_1 = \sqcup\{\Theta_i \mid i \in \{1, 2\}\}$; finally, for *3.* take partitions $\{1\} \cup \{2, 3\}$ and $\{1, 2\} \cup \{3\}$ of $I = \{1, 2, 3\}$, getting $\Theta_1 \sqcup (\Theta_2 \sqcup \Theta_3) = (\Theta_1 \sqcup \Theta_2) \sqcup \Theta_3 = \sqcup\{\Theta_i \mid i \in \{1, 2, 3\}\}$. □

Least Upper Bound Closures

We next define lub closures of sets of partial maps, a crucial operation for the the algorithms discussed next in the paper.

Definition 13 $\Theta \subseteq [X \rightharpoonup V]$ *is* ***lub closed*** *iff* $\sqcup\Theta' \in \Theta$ *for any* $\Theta' \subseteq \Theta$ *admitting upper bounds.*

Proposition 6 $\{\bot\}$ *and* $\{\bot, \theta\}$ *are lub closed (*$\theta \in [X \rightharpoonup V]$*).*

Proof: It follows easily from Definition 13, using the facts that $\sqcup\emptyset = \bot$ (*1.* in Proposition 3), $\sqcup\{\theta\} = \theta$ (*2.* in Proposition 3), and $\sqcup\{\bot, \theta\} = \theta$ (*3.* in Proposition 3 for $\Theta = \{\theta\}$). □

Proposition 7 *If* $\Theta \subseteq [X \rightharpoonup V]$ *and* $\{\Theta_i\}_{i \in I}$ *is a family of sets of partial functions in* $[X \rightharpoonup V]$*, then:*

1. *If* Θ *is lub closed then* $\bot \in \Theta$*; in particular,* $\emptyset$ *is not lub closed;*
2. *If* Θ *has upper bounds and is lub closed then it has a maximum;*
3. Θ *is lub closed iff* $\sqcup\{\Theta \mid i \in I\} = \Theta$ *for any* I*;*
4. *If* Θ *is lub closed and* $\Theta_i \subseteq \Theta$ *for each* $i \in I$ *then* $\sqcup\{\Theta_i \mid i \in I\} \subseteq \Theta$*;*

5. *If* Θ_i *is lub closed for each* $i \in I$ *then* $\sqcup\{\Theta_i \mid i \in I\}$ *is lub closed and* $\cup\{\Theta_i \mid i \in I\} \subseteq \sqcup\{\Theta_i \mid i \in I\}$;

6. *If* I *finite and* Θ_i *finite for all* $i \in I$, *then* $\sqcup\{\Theta_i \mid i \in I\}$ *finite;*

7. *If* Θ_i *lub closed for all* $i \in I$ *then* $\cap\{\Theta_i \mid i \in I\}$ *is lub closed;*

8. $\cap\{\Theta' \mid \Theta' \subseteq [X \rightharpoonup V]$ *with* $\Theta \subseteq \Theta'$ *and* Θ' *is lub closed*$\}$ *is the smallest lub closed set including* Θ.

Proof: *1.* follows taking $\Theta' = \emptyset$ in Definition 13 and using $\sqcup\emptyset = \bot$ (*1.* in Proposition 3).

2. follows taking $\Theta' = \Theta$ in Definition 13: $\sqcup\Theta \in \Theta$, so max Θ exists (and equals $\sqcup\Theta$).

3. Definition 12 implies that $\sqcup\{\Theta \mid i \in I\}$ equals $\{\sqcup\{\theta_i \mid i \in I\} \mid \theta_i \in \Theta$ for each $i \in I$, such that $\sqcup\{\theta_i \mid i \in I\}$ exists$\}$, which is nothing but $\{\sqcup\Theta' \mid \Theta' \subseteq \Theta$ such that $\sqcup\Theta'$ exists$\}$; the later can be now shown equal to Θ by double inclusion: $\{\sqcup\Theta' \mid \Theta' \subseteq \Theta$ such that $\sqcup\Theta'$ exists$\} \subseteq \Theta$ because Θ is lub closed, and $\Theta \subseteq \{\sqcup\Theta' \mid \Theta' \subseteq \Theta$ such that $\sqcup\Theta'$ exists$\}$ because one can pick $\Theta' = \{\theta\}$ for each $\theta \in \Theta$ and use the fact that $\sqcup\{\theta\} = \theta$ (*2.* in Proposition 3).

4. Let θ be an arbitrary partial function in $\sqcup\{\Theta_i \mid i \in I\}$, that is, $\theta = \sqcup\{\theta_i \mid i \in I\}$ for some $\theta_i \in \Theta_i$, one for each $i \in I$, such that $\{\theta_i \mid i \in I\}$ has upper bounds. Since Θ is lub closed and $\Theta_i \subseteq \Theta$ for each $i \in I$, it follows that $\theta \in \Theta$. Therefore, $\sqcup\{\Theta_i \mid i \in I\} \subseteq \Theta$.

5. Let Θ' be a set of partial functions included in $\sqcup\{\Theta_i \mid i \in I\}$ which admits an upper bound; moreover, for each $\theta' \in \Theta'$, let us fix a set $\{\theta_i^{\theta'} \mid i \in I\}$ such that $\theta_i^{\theta'} \in \Theta_i$ for each $i \in I$ and $\theta' = \sqcup\{\theta_i^{\theta'} \mid i \in I\}$ (such sets exist because $\theta' \in \Theta' \subseteq \sqcup\{\Theta_i \mid i \in I\}$). Let $\Theta^{\theta'}$ be the set $\{\theta_i^{\theta'} \mid i \in I\}$ for each $\theta' \in \Theta'$, let Θ_i' be the set $\{\theta_i^{\theta'} \mid \theta' \in \Theta'\}$ for each $i \in I$, and let Θ be the set $\{\theta_i^{\theta'} \mid \theta' \in \Theta', i \in I^{\theta'}\}$. It is easy to see that $\Theta = \cup\{\Theta^{\theta'} \mid \theta' \in \Theta'\} = \cup\{\Theta_i' \mid i \in I\}$ and that $\Theta_i' \subseteq \Theta_i$ for each $i \in I$. Since $\sqcup\Theta'$ exists (because Θ' has upper bounds) and $\sqcup\Theta' = \sqcup\{\theta' \mid \theta' \in \Theta'\} = \sqcup\{\sqcup\Theta^{\theta'} \mid \theta' \in \Theta'\}$, by *5.* in Proposition 3 it follows that $\sqcup\Theta$ exists and $\sqcup\Theta' = \sqcup\Theta$. Since $\Theta = \cup\{\Theta_i' \mid i \in I\}$ and $\sqcup\Theta$ exists, by *5.* in Proposition 3 again we get that $\sqcup\{\sqcup\Theta_i' \mid i \in I\}$ exists and is equal to $\sqcup\Theta$, which is equal to $\sqcup\Theta'$. Since $\Theta_i' \subseteq \Theta_i$ and Θ_i is lub closed, we get that $\sqcup\Theta_i' \in \Theta_i$. That means that $\sqcup\{\sqcup\Theta_i' \mid i \in I\} \in \sqcup\{\Theta_i \mid i \in I\}$, that is, that $\sqcup\Theta' \in \sqcup\{\Theta_i \mid i \in I\}$. Since $\Theta' \subseteq \sqcup\{\Theta_i \mid i \in I\}$ was chosen arbitrarily, we conclude that $\sqcup\{\Theta_i \mid i \in I\}$ is lub closed. To show that $\cup\{\Theta_i \mid i \in I\} \subseteq \sqcup\{\Theta_i \mid i \in I\}$, let us pick an $i \in I$ and let us partition I as $\{i\} \cup (I \backslash \{i\})$. By Proposition 5, $\sqcup\{\Theta_i \mid i \in I\} = \Theta_i \sqcup (\sqcup\{\Theta_j \mid j \in I \backslash \{i\}\})$. The proof above also implies that $\sqcup\{\Theta_j \mid j \in I \backslash \{i\}\}$ is lub closed, so by *1.* we get that $\bot \in \sqcup\{\Theta_j \mid j \in I \backslash \{i\}\}$. Finally, *6.* in

Proposition 4 implies $\Theta_i \sqsubseteq \Theta_i \sqcup (\sqcup\{\Theta_j \mid j \in I\backslash\{i\}\})$, so $\Theta_i \subseteq \sqcup\{\Theta_i \mid i \in I\}$ for each $i \in I$, that is, $\cup\{\Theta_i \mid i \in I\} \subseteq \sqcup\{\Theta_i \mid i \in I\}$.

6. Recall from Definition 12 that $\sqcup\{\Theta_i \mid i \in I\}$ contains the existing least upper bounds of sets of partial functions containing precisely one partial function in each Θ_i. If I and each of the Θ_i for each $i \in I$ is finite, then $|\sqcup\{\Theta_i \mid i \in I\}| \leq \prod_{i\in I} |\Theta_i|$, because there at most $\prod_{i\in I} |\Theta_i|$ combinations of partial functions, one in each Θ_i, that admit an upper bound. Therefore, $\sqcup\{\Theta_i \mid i \in I\}$ is also finite.

7. Let $\Theta' \subseteq \cap\{\Theta_i \mid i \in I\}$ be a set of partial functions admitting an upper bound. Then $\Theta' \subseteq \Theta_i$ for each $i \in I$ and, since each Θ_i is lub closed, $\sqcup\Theta' \in \Theta_i$ for each $i \in I$. Therefore, $\sqcup\Theta' \in \cap\{\Theta_i \mid i \in I\}$.

8. Anticipating the definition of and notation for lub closures (Definition 13), we let $\overline{\Theta}$ denote the set $\cap\{\Theta' \mid \Theta' \subseteq [X \rightharpoonup V]$ with $\Theta \subseteq \Theta'$ and Θ' is lub closed$\}$. It is clear that $\Theta \subseteq \overline{\Theta}$ and, by *7.*, that $\overline{\Theta}$ is lub closed. It is also the *smallest* lub closed set including Θ, because all such sets Θ' are among those whose intersection defines $\overline{\Theta}$. □

Definition 14 *Given $\theta' \in [X \rightharpoonup V]$ and $\Theta \subseteq [X \rightharpoonup V]$, let*

$$(\theta']_\Theta \stackrel{\text{def}}{=} \{\theta \mid \theta \in \Theta \text{ and } \theta \sqsubseteq \theta'\}$$

be the set of partial functions in Θ that are less informative than θ'.

Proposition 8 *If $\theta, \theta', \theta'', \theta_1, \theta_2 \in [X \rightharpoonup V]$ and if $\Theta \subseteq [X \rightharpoonup V]$ is lub closed, then:*

1. *$(\theta']_\Theta$ is lub closed and $\max(\theta']_\Theta$ exists;*
2. *If $\theta' \in \{\theta\} \sqcup \Theta$ then $\{\theta'' \mid \theta'' \in \Theta$ and $\theta' = \theta \sqcup \theta''\}$ has maximum and that equals $\max(\theta']_\Theta$;*
3. *If $\theta_1, \theta_2 \in \{\theta\} \sqcup \Theta$ such that $\theta_1 = \max(\theta_2]_\Theta$, then $\theta_1 = \theta_2$.*

Proof: *1.* First, note that θ' is an upper bound for $(\theta']_\Theta$ as well as for any subset Θ' of it, so any $\Theta' \subseteq (\theta']_\Theta$ has upper bounds, so by *2.* in Proposition 2, $\sqcup\Theta'$ exists for any $\Theta' \subseteq (\theta']_\Theta$. Moreover, if $\Theta' \subseteq (\theta']_\Theta$ then $\sqcup\Theta' \sqsubseteq \theta'$, and since Θ is lub closed it follows that $\sqcup\Theta' \in \Theta$, so $\sqcup\Theta' \in (\theta']_\Theta$. Therefore, $(\theta']_\Theta$ is lub closed. *2.* in Proposition 7 now implies that $(\theta']_\Theta$ has maximum; to be concrete, $\max(\theta']_\Theta$ is nothing but $\sqcup(\theta']_\Theta$, which belongs to $(\theta']_\Theta$ (because one can pick $\Theta' = (\theta']_\Theta$ above).

2. Let Q be the set of partial functions $\{\theta'' \mid \theta'' \in \Theta \text{ and } \theta' = \theta \sqcup \theta''\}$. Note that Q is non-empty (because $\theta' \in \{\theta\} \sqcup \Theta$, so there is some $\theta'' \in \Theta$ such as $\theta' = \theta \sqcup \theta''$) and has upper-bounds (because θ' is an upper bound for it), but that it is not necessarily lub closed (because, unless $\theta' = \theta$, Q does not contain $\bot$, contradicting *1.* in Proposition 7). Hence Q has a lub (by *2.* in Proposition 2), say q, and $q = \sqcup Q \sqsubseteq \theta'$; since $\theta \sqsubseteq \theta'$, it follows that $\theta \sqcup q \sqsubseteq \theta'$. On the other hand $\theta' \sqsubseteq \theta \sqcup q$ by *4.* in Proposition 3, because there is some $\theta'' \in Q$ such that $\theta' = \theta \sqcup \theta''$ and $\theta'' \sqsubseteq q$. Therefore, $\theta' = \theta \sqcup q$. Since Θ is lub closed, it follows that $q \in \Theta$. Therefore, $q \in Q$, so q is the maximum element of Q. Let us next show that $q = \max(\theta']_\Theta$. The relation $q \sqsubseteq \max(\theta']_\Theta$ is immediate because $q \in (\theta']_\Theta$ (we proved above that $q \in \Theta$ and $q \sqsubseteq \theta'$). For $\max(\theta']_\Theta \sqsubseteq q$ it suffices to show that $\max(\theta']_\Theta \in Q$, that is, that $\theta \sqcup \max(\theta']_\Theta = \theta'$: $\theta \sqcup \max(\theta']_\Theta \sqsubseteq \theta'$ follows because $\theta \sqsubseteq \theta'$ and $\max(\theta']_\Theta \sqsubseteq \theta'$, while $\theta' \sqsubseteq \theta \sqcup \max(\theta']_\Theta$ follows because there is some $\theta'' \in \Theta$ such that $\theta' = \theta \sqcup \theta''$ and, since $\theta'' \sqsubseteq \max(\theta']_\Theta$, $\theta \sqcup \theta'' \sqsubseteq \theta \sqcup \max(\theta']_\Theta$ (by *4.* in Proposition 3).

3. admits a direct proof simpler than that of *2.*; however, since *2.* is needed anyway, we prefer to use *2.* Note that $\theta \sqsubseteq \theta_1 \sqsubseteq \theta_2$. By *2.*, $\theta_1 = \max\{\theta'' \mid \theta'' \in \Theta \text{ and } \theta_2 = \theta \sqcup \theta''\}$, which implies $\theta_2 = \theta \sqcup \theta_1 = \theta_1$. □

Definition 15 *Given $\Theta \subseteq [X \rightharpoonup V]$, we let $\overline{\Theta}$, the* ***least upper bound (lub) closure*** *of Θ, be defined as follows:*

$$\overline{\Theta} \stackrel{\text{def}}{=} \cap\{\Theta' \mid \Theta' \subseteq [X \rightharpoonup V] \textit{ with } \Theta \subseteq \Theta' \textit{ and } \Theta' \textit{ is lub closed}\}.$$

Proposition 9 *The next hold ($\Theta \subseteq [X \rightharpoonup V], \theta \in [X \rightharpoonup V]$):*

1. *$\overline{\Theta}$ is the smallest lub closed set including Θ;*
2. *$\overline{\emptyset} = \overline{\{\bot\}} = \{\bot\}$;*
3. *$\overline{\{\theta\}} = \{\bot, \theta\}$.*

Proof: *1.* follows by *7.* in Proposition 7. For *2.* and *3.*, first note that $\{\bot\}$ and $\{\bot, \theta\}$ are lub closed by Proposition 6; second, note that they are indeed the smallest lub closed sets including $\bot$ and resp. θ, as any lub closed set must include $\bot$ (*1.* in Proposition 7). □

Proposition 10 *The lub closure map $\bar{\cdot} : 2^{[X \rightharpoonup V]} \to 2^{[X \rightharpoonup V]}$ is a closure operator, that is, for any $\Theta, \Theta_1, \Theta_2 \subseteq [X \rightharpoonup V]$,*

1. *(extensivity)* $\Theta \subseteq \overline{\Theta}$;

2. *(monotonicity) If* $\Theta_1 \subseteq \Theta_2$ *then* $\overline{\Theta_1} \subseteq \overline{\Theta_2}$;

3. *(idempotency)* $\overline{\overline{\Theta}} = \overline{\Theta}$.

Proof: Extensivity and idempotency follow immediately from the definitions of $\overline{\Theta}$ and $\overline{\overline{\Theta}}$ (which are lub closed by *1.* in Proposition 9). For monotonicity, one should note that $\overline{\Theta_2}$ satisfies the properties of $\overline{\Theta_1}$ (i.e., $\Theta_1 \subseteq \overline{\Theta_2}$ and Θ_2 is lub closed); since $\overline{\Theta_1}$ is the smallest with those properties, it follows that $\overline{\Theta_1} \subseteq \overline{\Theta_2}$. □

Proposition 11 $\overline{\cup\{\Theta_i \mid i \in I\}} = \sqcup\{\overline{\Theta_i} \mid i \in I\}$ *for any family* $\{\Theta_i\}_{i\in I}$ *of partial functions in* $[X \rightharpoonup V]$.

Proof: Since $\overline{\Theta_i}$ is lub closed for any $i \in I$, *5.* in Proposition 7 implies that $\sqcup\{\overline{\Theta_i} \mid i \in I\}$ is lub closed and $\cup\{\overline{\Theta_i} \mid i \in I\} \subseteq \sqcup\{\overline{\Theta_i} \mid i \in I\}$. Since *1.* in Proposition 10 implies $\Theta_i \subseteq \overline{\Theta_i}$ for each $i \in I$ and since $\overline{\cup\{\Theta_i \mid i \in I\}}$ is the smallest lub closed set including $\cup\{\Theta_i \mid i \in I\}$ (*1.* in Proposition 9), the inclusion $\overline{\cup\{\Theta_i \mid i \in I\}} \subseteq \sqcup\{\overline{\Theta_i} \mid i \in I\}$ holds. Conversely, *2.* in Proposition 10 implies that $\overline{\Theta_i} \subseteq \overline{\cup\{\Theta_i \mid i \in I\}}$ for any $i \in I$, so $\sqcup\{\overline{\Theta_i} \mid i \in I\} \subseteq \overline{\cup\{\Theta_i \mid i \in I\}}$ holds by *4.* in Proposition 7. □

Corollary 2 *For any* $\theta \in [X \rightharpoonup V]$ *and any* $\Theta \subseteq [X \rightharpoonup V]$, *the equality* $\overline{\{\theta\} \cup \Theta} = \{\bot, \theta\} \sqcup \overline{\Theta}$ *holds.*

Proof: $\overline{\{\theta\} \cup \Theta} = \overline{\{\theta\}} \sqcup \overline{\Theta}$ by Proposition 11, and further $\overline{\{\theta\}} = \{\bot, \theta\}$ by *3.* in Proposition 9. □

Corollary 3 *If* $\Theta \subseteq [X \rightharpoonup V]$ *is finite then* $\overline{\Theta}$ *is also finite.*

Proof: Suppose that $\Theta = \{\theta_1, \theta_2, \ldots, \theta_n\}$ for some $n \geq 0$. Iteratively applying Corollary 2, $\overline{\Theta} = \{\bot, \theta_1\} \sqcup \{\bot, \theta_2\} \sqcup \cdots \{\bot, \theta_n\}$; in obtaining that, we used *2.* in Proposition 9 and *1.* in Corollary 1. The result follows now by *6.* in Proposition 7. □

4.2.3 Slicing with Less

Consider a parametric trace τ in $\mathcal{E}\langle X\rangle^*$ and a parameter instance θ. Since there is no apriori correlation between the parameters being instantiated by θ and those by the various parametric events in τ, it may very well be the case that θ contains parameter instances that never appear in τ. In this section we show that slicing τ by θ is the same as slicing it by a "smaller" parameter

instance than θ, namely one containing only those parameters instantiated by θ that also appear as instances of some parameters in some events in τ. Formally, this smaller parameter instance is the largest partial map smaller than θ in the lub closure of all the parameter instances of events in τ; this partial function is proved to indeed exist. We first formalize a notation used informally so far in this paper:

Notation 1 *When the domain of θ is finite, which is always the case in our examples in this paper and probably will also be the case in most practical uses of our trace slicing algorithm, and when the corresponding parameter names are clear from context, we take the liberty to write partial functions compactly by simply listing their parameter values; for example, we write a partial function θ with $\theta(a) = a_2$, $\theta(b) = b_1$ and $\theta(c) = c_1$ as the sequence "$a_2b_1c_1$". The function $\perp$ then corresponds to the empty sequence.*

Example. Here is a parametric trace with events parametric in $\{a, b, c\}$ taking values in $\{a_1, a_2, b_1, c_1\}$:
$\tau = e_1\langle a_1\rangle\ e_2\langle a_2\rangle\ e_3\langle b_1\rangle$
$e_4\langle a_2b_1\rangle\ e_5\langle a_1\rangle\ e_6\langle\rangle\ e_7\langle b_1\rangle\ e_8\langle c_1\rangle\ e_9\langle a_2c_1\rangle\ e_{10}\langle a_1b_1c_1\rangle\ e_{11}\langle\rangle$. It may be the case that some of the base events appearing in a trace are the same; for example, e_1 may be equal to e_2 and to e_5. It is actually frequently the case in practice (at least in PQL [66], Tracematches [3], JavaMOP [29]) that parametric events are specified apriori with a given (sub)set of parameters, so that each event in $\mathcal{E}$ is always instantiated with partial functions over the same domain, that is, if $e\langle\theta\rangle$ and $e\langle\theta'\rangle$ appear in some parametric trace, then $Dom(\theta) = Dom(\theta')$. While this restriction is reasonable, our trace slicing and monitoring algorithms do not need it. □

Recall from Definition 7 that the trace slice $\tau{\upharpoonright_\theta}$ keeps from τ only those events that are relevant for θ and drops their parameters.

Example. Consider again the sample parametric trace above with events parametric in $\{a, b, c\}$:
$\tau = e_1\langle a_1\rangle\ e_2\langle a_2\rangle\ e_3\langle b_1\rangle\ e_4\langle a_2b_1\rangle$
$e_5\langle a_1\rangle\ e_6\langle\rangle\ e_7\langle b_1\rangle\ e_8\langle c_1\rangle\ e_9\langle a_2c_1\rangle\ e_{10}\langle a_1b_1c_1\rangle\ e_{11}\langle\rangle$. Several slices of τ are listed below:

$$
\begin{aligned}
\tau\!\upharpoonright_{a_1} &= e_1e_5e_6e_{11}\\
\tau\!\upharpoonright_{a_2} &= e_2e_6e_{11}\\
\tau\!\upharpoonright_{a_1b_1} &= e_1e_3e_5e_6e_7e_{11}\\
\tau\!\upharpoonright_{a_2b_1} &= e_2e_3e_4e_6e_7e_{11}\\
\tau\!\upharpoonright &= e_6e_{11}\\
\tau\!\upharpoonright_{a_1b_1c_1} &= e_1e_3e_5e_6e_7e_8e_{10}e_{11}\\
\tau\!\upharpoonright_{a_2b_1c_1} &= e_2e_3e_4e_6e_7e_8e_9e_{11}\\
\tau\!\upharpoonright_{a_1b_2c_1} &= e_1e_5e_6e_8e_{11}\\
\tau\!\upharpoonright_{b_2c_2} &= e_6e_{11}
\end{aligned}
$$

In order for the partial functions above to make sense, we assumed that the set V_X in which parameters $X = \{a, b, c\}$ take values includes $\{a_1, a_2, b_1, b_2, c_1, c_2\}$. □

Definition 16 *Given parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$, we let Θ_τ denote the lub closure of all the parameter instances appearing in events in τ, that is, $\Theta_\tau = \overline{\{\theta \mid \theta \in [X \rightarrow V],\ e\langle\theta\rangle \in \tau\}}$.*

Proposition 12 *Θ_τ is a finite lub closed set for any $\tau \in \mathcal{E}\langle X\rangle^*$.*

Proof: Θ_τ is already defined as a lub closed set; since τ is finite, Corollary 3 implies that Θ_τ is finite. □

Proposition 13 *Given $\tau\, e\langle\theta\rangle \in \mathcal{E}\langle X\rangle^*$, the following equality holds: $\Theta_{\tau\, e\langle\theta\rangle} = \{\bot, \theta\} \sqcup \Theta_\tau$.*

Proof: It follows by the following sequence of equalities:

$$
\begin{aligned}
\Theta_{\tau\, e\langle\theta\rangle} &= \overline{\{\theta' \mid \theta' \in [X \rightarrow V],\ e'\langle\theta'\rangle \in \tau\, e\langle\theta\rangle\}}\\
&= \overline{\{\theta\} \cup \{\theta' \mid \theta' \in [X \rightarrow V],\ e'\langle\theta'\rangle \in \tau\}}\\
&= \overline{\{\theta\} \cup \Theta_\tau}\\
&= \{\bot, \theta\} \sqcup \overline{\Theta_\tau}\\
&= \{\bot, \theta\} \sqcup \Theta_\tau.
\end{aligned}
$$

The first equality follows by Definition 16, the second by separating the case $e'\langle\theta'\rangle = e\langle\theta\rangle$, the third again by Definition 16, the fourth by Corollary 2, and the fifth by Proposition 12. Therefore, $\Theta_{\tau\, e\langle\theta\rangle}$ is the smallest lub closed set that contains θ and includes Θ_τ. □

Proposition 14 *Given $\tau \in \mathcal{E}\langle X\rangle^*$ and $\theta \in [X \rightarrow V]$, the following equality holds: $\tau\!\upharpoonright_\theta = \tau\!\upharpoonright_{\max(\theta]_{\Theta_\tau}}$.*

Proof: We prove the following more general result:

"let $\Theta \subseteq [X \rightarrow V]$ be lub closed and let $\theta \in [X \rightarrow V]$;

then $\tau\restriction_\theta = \tau\restriction_{\max(\theta]_\Theta}$ for any $\tau \in \mathcal{E}\langle X\rangle^*$ with $\Theta_\tau \subseteq \Theta$."

First note that the statement above is well-formed because $\max(\theta]_\Theta$ exists whenever Θ is lub closed (*1.* in Proposition 8), and that it is indeed more general than the stated result: for the given $\tau \in \mathcal{E}\langle X\rangle^*$ and $\theta \in [X \rightarrow V]$, we pick Θ to be Θ_τ. We prove the general result by induction on the *length* of τ:

- If $|\tau| = 0$ then $\tau = \epsilon$ and $\epsilon\restriction_\theta = \epsilon\restriction_{\max(\theta]_\Theta} = \epsilon$.

- Now suppose that $\tau\restriction_\theta = \tau\restriction_{\max(\theta]_\Theta}$ for any $\tau \in \mathcal{E}\langle X\rangle^*$ with $\Theta_\tau \subseteq \Theta$ and $|\tau| = n \geq 0$, and let us show that $\tau'\restriction_\theta = \tau'\restriction_{\max(\theta]_\Theta}$ for any $\tau' \in \mathcal{E}\langle X\rangle^*$ with $\Theta_{\tau'} \subseteq \Theta$ and $|\tau'| = n+1$. Pick such a τ' and let $\tau' = \tau\, e\langle\theta'\rangle$ for a $\tau \in \mathcal{E}\langle X\rangle^*$ with $|\tau| = n$ and an $e\langle\theta'\rangle \in \mathcal{E}\langle X\rangle$. Since $\Theta_{\tau'} \subseteq \Theta$, by *6.* in Proposition 4 and by Proposition 13 it follows that $\Theta_\tau \subseteq \{\bot, \theta'\} \sqcup \Theta_\tau \subseteq \Theta$, so the induction hypothesis implies $\tau\restriction_\theta = \tau\restriction_{\max(\theta]_\Theta}$. The rest follows noticing that $\theta' \sqsubseteq \theta$ iff $\theta' \sqsubseteq \max(\theta]_\Theta$, which is a consequence of the definition of $\max(\theta]_\Theta$ because $\theta' \in \{\bot, \theta'\} \subseteq \{\bot, \theta'\} \sqcup \Theta_\tau \subseteq \Theta$ (again by *6.* in Proposition 4 and by Proposition 13).

Alternatively, one could have also done the proof above by induction on τ, not on its length, but the proof would be more involved, because one would need to prove that the domain over which the property is universally quantified, namely "any $\tau \in \mathcal{E}\langle X\rangle^*$ with $\Theta_\tau \subseteq \Theta$" is inductively generated. We therefore preferred to choose a more elementary induction schema. □

4.2.4 Parametric Trace Slicing Algorithm $\mathbb{A}\langle X\rangle$

We next define an algorithm $\mathbb{A}\langle X\rangle$ that takes a parametric trace $\tau \in \mathcal{E}\langle X\rangle^*$ incrementally (i.e., event by event), and builds a partial function $\mathbb{T} \in [[X \rightarrow V] \rightarrow \mathcal{E}^*]$ of finite domain that serves as a quick lookup table for all slices of τ. More precisely, Theorem 1 shows that, for any $\theta \in [X \rightarrow V]$, the trace slice $\tau\restriction_\theta$ is $\mathbb{T}(\max(\theta]_\Theta)$ after $\mathbb{A}\langle X\rangle$ processes τ, where $\Theta = \Theta_\tau$ is the domain of $\mathbb{T}$, a finite lub closed set of partial functions also calculated by $\mathbb{A}\langle X\rangle$ incrementally. Therefore, assuming that $\mathbb{A}\langle X\rangle$ is run on trace τ, all one has to do in order to calculate a slice $\tau\restriction_\theta$ for a given $\theta \in [X \rightarrow V]$ is to calculate $\max(\theta]_\Theta$ followed by a lookup into $\mathbb{T}$. This way the trace τ, which can be very long, is processed/traversed only once, as it is being generated, and appropriate data-structures are maintained by our algorithm that allow for retrieval of slices for any parameter instance θ, without having to traverse the trace τ again, as an algorithm blindly following the definition of trace slicing would do.

```
Algorithm 𝔸⟨X⟩
Input: parametric trace τ ∈ ℰ⟨X⟩*
Output: map 𝕋 ∈ [[X→V]→ℰ*] and set Θ ⊆ [X→V]

1  𝕋 ← ⊥; 𝕋(⊥) ← ϵ; Θ ← {⊥}
2  for all parametric event e⟨θ⟩ in order (fist to last) in τ do
3  ⋮ for all θ′ ∈ {θ} ⊔ Θ do
4  ⋮ ⋮ 𝕋(θ′) ← 𝕋(max(θ′)_Θ) e
5  ⋮ endfor
6  ⋮ Θ ← {⊥, θ} ⊔ Θ
7  endfor
```

Figure 4.4: Parametric trace slicing algorithm $\mathbb{A}\langle X\rangle$.

Figure 4.4 shows our trace slicing algorithm $\mathbb{A}\langle X\rangle$. In spite of $\mathbb{A}\langle X\rangle$'s small size, its proof of correctness is surprisingly intricate, making use of almost all the mathematical machinery developed so far in the paper. The algorithm $\mathbb{A}\langle X\rangle$ on input τ, written more succinctly $\mathbb{A}\langle X\rangle(\tau)$, traverses τ from its first event to its last event and, for each encountered event $e\langle\theta\rangle$, updates both its data-structures, $\mathbb{T}$ and Θ. After processing each event, the relationship between $\mathbb{T}$ and Θ is that the latter is the domain of the former. Line 1 initializes the data-structures: $\mathbb{T}$ is undefined everywhere (i.e., $\bot$) except for the undefined-everywhere function $\bot$, where $\mathbb{T}(\bot) = \epsilon$; as expected, Θ is then initialized to the set $\{\bot\}$. The code (lines 3 to 6) inside the outer loop (lines 2 to 7) can be triggered when a new event is received, as in most online runtime verification systems. When a new event is received, say $e\langle\theta\rangle$, the mapping $\mathbb{T}$ is updated as follows: for each $\theta' \in [X \rightarrow V]$ that can be obtained by combining θ with the compatible partial functions in the domain of the current $\mathbb{T}$, update $\mathbb{T}(\theta')$ by adding the non-parametric event e to the end of the slice corresponding to the largest (i.e., most "knowledgeable") entry in the current table $\mathbb{T}$ that is less informative or as informative as θ'; the Θ data-structure is then extended in line 6.

Example.

Consider again the sample parametric trace above with events parametric in $\{a, b, c\}$, namely $\tau = e_1\langle a_1\rangle\ e_2\langle a_2\rangle\ e_3\langle b_1\rangle$
$e_4\langle a_2 b_1\rangle\ e_5\langle a_1\rangle\ e_6\langle\rangle\ e_7\langle b_1\rangle\ e_8\langle c_1\rangle\ e_9\langle a_2 c_1\rangle\ e_{10}\langle a_1 b_1 c_1\rangle\ e_{11}\langle\rangle$. Table 4.2.4 shows how $\mathbb{A}\langle X\rangle$ works on τ. An entry of the form "$\langle\theta\rangle : w$" in a table cell corresponding to a "current" parametric event $e\langle\theta\rangle$ means that $\mathbb{T}(\theta) = w$ after processing all the parametric events up to and including the current one; $\mathbb{T}$ is undefined on any other partial function. Obviously, the Θ corresponding to a cell is the union of all the θ's that appear in pairs "$\langle\theta\rangle : w$" in that cell. Note that, as each parametric event $e\langle\theta\rangle$ is processed, the non-parametric event e is added at most once to each slice, and that the Θ

$e_1\langle a_1\rangle$	$e_2\langle a_2\rangle$	$e_3\langle b_1\rangle$	$e_4\langle a_2b_1\rangle$
$\langle\rangle : \epsilon$	$\langle\rangle : \epsilon$	$\langle\rangle : \epsilon$	$\langle\rangle : \epsilon$
$\langle a_1\rangle : e_1$	$\langle a_1\rangle : e_1$	$\langle a_1\rangle : e_1$	$\langle a_1\rangle : e_1$
	$\langle a_2\rangle : e_2$	$\langle a_2\rangle : e_2$	$\langle a_2\rangle : e_2$
		$\langle b_1\rangle : e_3$	$\langle b_1\rangle : e_3$
		$\langle a_1b_1\rangle : e_1e_3$	$\langle a_1b_1\rangle : e_1e_3$
		$\langle a_2b_1\rangle : e_2e_3$	$\langle a_2b_1\rangle : e_2e_3e_4$

$e_5\langle a_1\rangle$	$e_6\langle\rangle$	$e_7\langle b_1\rangle$
$\langle\rangle : \epsilon$	$\langle\rangle : e_6$	$\langle\rangle : e_6$
$\langle a_1\rangle : e_1e_5$	$\langle a_1\rangle : e_1e_5e_6$	$\langle a_1\rangle : e_1e_5e_6$
$\langle a_2\rangle : e_2$	$\langle a_2\rangle : e_2e_6$	$\langle a_2\rangle : e_2e_6$
$\langle b_1\rangle : e_3$	$\langle b_1\rangle : e_3e_6$	$\langle b_1\rangle : e_3e_6e_7$
$\langle a_1b_1\rangle : e_1e_3e_5$	$\langle a_1b_1\rangle : e_1e_3e_5e_6$	$\langle a_1b_1\rangle : e_1e_3e_5e_6e_7$
$\langle a_2b_1\rangle : e_2e_3e_4$	$\langle a_2b_1\rangle : e_2e_3e_4e_6$	$\langle a_2b_1\rangle : e_2e_3e_4e_6e_7$

$e_8\langle c_1\rangle$	$e_9\langle a_2c_1\rangle$	$e_{10}\langle a_1b_1c_1\rangle$	$e_{11}\langle\rangle$
$\langle\rangle : e_6$	$\langle\rangle : e_6$	$\langle\rangle : e_6$	$\langle\rangle : e_6e_{11}$
$\langle a_1\rangle : e_1e_5e_6$	$\langle a_1\rangle : e_1e_5e_6$	$\langle a_1\rangle : e_1e_5e_6$	$\langle a_1\rangle : e_1e_5e_6e_{11}$
$\langle a_2\rangle : e_2e_6$	$\langle a_2\rangle : e_2e_6$	$\langle a_2\rangle : e_2e_6$	$\langle a_2\rangle : e_2e_6e_{11}$
$\langle b_1\rangle : e_3e_6e_7$	$\langle b_1\rangle : e_3e_6e_7$	$\langle b_1\rangle : e_3e_6e_7$	$\langle b_1\rangle : e_3e_6e_7e_{11}$
$\langle a_1b_1\rangle : e_1e_3e_5e_6e_7$	$\langle a_1b_1\rangle : e_1e_3e_5e_6e_7$	$\langle a_1b_1\rangle : e_1e_3e_5e_6e_7$	$\langle a_1b_1\rangle : e_1e_3e_5e_6e_7e_{11}$
$\langle a_2b_1\rangle : e_2e_3e_4e_6e_7$	$\langle a_2b_1\rangle : e_2e_3e_4e_6e_7$	$\langle a_2b_1\rangle : e_2e_3e_4e_6e_7$	$\langle a_2b_1\rangle : e_2e_3e_4e_6e_7e_{11}$
$\langle c_1\rangle : e_6e_8$	$\langle c_1\rangle : e_6e_8$	$\langle c_1\rangle : e_6e_8$	$\langle c_1\rangle : e_6e_8e_{11}$
$\langle a_1c_1\rangle : e_1e_5e_6e_8$	$\langle a_1c_1\rangle : e_1e_5e_6e_8$	$\langle a_1c_1\rangle : e_1e_5e_6e_8$	$\langle a_1c_1\rangle : e_1e_5e_6e_8e_{11}$
$\langle a_2c_1\rangle : e_2e_6e_8$	$\langle a_2c_1\rangle : e_2e_6e_8e_9$	$\langle a_2c_1\rangle : e_2e_6e_8e_9$	$\langle a_2c_1\rangle : e_2e_6e_8e_9e_{11}$
$\langle b_1c_1\rangle : e_3e_6e_7e_8$	$\langle b_1c_1\rangle : e_3e_6e_7e_8$	$\langle b_1c_1\rangle : e_3e_6e_7e_8$	$\langle b_1c_1\rangle : e_3e_6e_7e_8e_{11}$
$\langle a_1b_1c_1\rangle : e_1e_3e_5e_6e_7e_8$	$\langle a_1b_1c_1\rangle : e_1e_3e_5e_6e_7e_8$	$\langle a_1b_1c_1\rangle : e_1e_3e_5e_6e_7e_8e_{10}$	$\langle a_1b_1c_1\rangle : e_1e_3e_5e_6e_7e_8e_{10}e_{11}$
$\langle a_2b_1c_1\rangle : e_2e_3e_4e_6e_7e_8$	$\langle a_2b_1c_1\rangle : e_2e_3e_4e_6e_7e_8e_9$	$\langle a_2b_1c_1\rangle : e_2e_3e_4e_6e_7e_8e_9$	$\langle a_2b_1c_1\rangle : e_2e_3e_4e_6e_7e_8e_9e_{11}$

Table 4.1: A run of the trace slicing algorithm $\mathbb{A}\langle X\rangle$ (top-left table first, followed by bottom-left table, followed by the right table).

corresponding to each cell is lub closed. □

$\mathbb{A}\langle X\rangle$ compactly and uniformly captures several special cases and subcases that are worth discussing. The discussion below can be formalized as an inductive (on the length of τ) proof of correctness for $\mathbb{A}\langle X\rangle$, but we prefer to keep this discussion informal and give a rigorous proof shortly after. The role of this discussion is twofold: (1) to better explain the algorithm $\mathbb{A}\langle X\rangle$, providing the reader with additional intuition for its difficulty and compactness, and (2) to give a proof sketch for the correctness of $\mathbb{A}\langle X\rangle$.

Let us first note that a partial function added to Θ will never be removed from Θ; that's because $\Theta \subseteq \{\bot, \theta\} \sqcup \Theta$. The same holds true for the domain of $\mathbb{T}$, because line 4 can only add new elements to $Dom(\mathbb{T})$; in fact, the domain of $\mathbb{T}$ is extended with precisely the set $\{\theta\} \sqcup \Theta$ after each event parametric in θ is processed by $\mathbb{A}\langle X\rangle$. Moreover, since $Dom(\mathbb{T}) = \Theta = \Theta_\epsilon = \{\bot\}$ initially and since *5.* and *7.* in Proposition 4 imply $\Theta \cup (\{\theta\} \sqcup \Theta) = \{\bot, \theta\} \sqcup \Theta$ while Proposition 13 states that $\Theta_{\tau\, e\langle\theta\rangle} = \{\bot, \theta\} \sqcup \Theta_\tau$, we can inductively show that $Dom(\mathbb{T}) = \Theta = \Theta_\tau$ each time after $\mathbb{A}\langle X\rangle$ is executed on a parametric trace τ.

Each θ' considered by the loop at lines 3-5 has the property that $\theta \sqsubseteq \theta'$, and at (precisely) one iteration of the loop θ' is θ; indeed, $\theta \in \{\theta\} \sqcup \Theta$ because $\bot \in \Theta$. Thanks to Proposition 14, the claimed Theorem 1 holds essentially iff $\mathbb{T}(\theta') = \tau{\restriction_{\theta'}}$ after $\mathbb{T}(\theta')$ is updated in line 4. A tricky observation which is crucial for this is that *3.* in Proposition 8 implies that the updates of $\mathbb{T}(\theta')$ do

not interfere with each other for different $\theta' \in \{\theta\} \sqcup \Theta$; otherwise the non-parametric event e may be added multiple times to some trace slices $\mathbb{T}(\theta')$.

Let us next informally argue, inductively, that it is indeed the case that $\mathbb{T}(\theta') = \tau{\restriction_{\theta'}}$ after $\mathbb{T}(\theta')$ is updated in line 4 (it vacuously holds on the empty trace). Since $\max(\theta']_\Theta \in \Theta$, the inductive hypothesis tells us that $\mathbb{T}(\max(\theta']_\Theta) = \tau{\restriction_{\max(\theta']_\Theta}}$; these are further equal to $\tau{\restriction_{\theta'}}$ by Proposition 14. Since $\theta \sqsubseteq \theta'$, the definition of trace slicing implies that $(\tau\, e\langle\theta\rangle){\restriction_{\theta'}} = \tau{\restriction_{\theta'}}\ e$. Therefore, $\mathbb{T}(\theta')$ is indeed $(\tau\, e\langle\theta\rangle){\restriction_{\theta'}}$ after line 4 of $\mathbb{A}\langle X\rangle$ is executed while processing the event $e\langle\theta\rangle$ that follows trace τ. This concludes our informal proof sketch; let us next give a rigorous proof of correctness for our trace slicing algorithm.

Definition 17 *Let $\mathbb{A}\langle X\rangle(\tau).\mathbb{T}$ and $\mathbb{A}\langle X\rangle(\tau).\Theta$ be the two data-structures ($\mathbb{T}$ and Θ) of $\mathbb{A}\langle X\rangle$ after it processes τ.*

Theorem 1 *The following hold for any $\tau \in \mathcal{E}\langle X\rangle^*$:*

1. *$Dom(\mathbb{A}\langle X\rangle(\tau).\mathbb{T}) = \mathbb{A}\langle X\rangle(\tau).\Theta = \Theta_\tau$;*
2. *$\mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\theta) = \tau{\restriction_\theta}$ for any $\theta \in \mathbb{A}\langle X\rangle(\tau).\Theta$;*
3. *$\tau{\restriction_\theta} = \mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\max(\theta]_{\mathbb{A}\langle X\rangle(\tau).\Theta})$ for any $\theta \in [X \rightharpoonup V]$.*

Proof: Since $\mathbb{A}\langle X\rangle$ processes the events in the input trace in order, when given the input $\tau\, e\langle\theta\rangle$, the Θ and $\mathbb{T}$ structures after $\mathbb{A}\langle X\rangle$ processes τ but before it processes $e\langle\theta\rangle$ (i.e., right before the last iteration of the loop at lines 2-7) are precisely $\mathbb{A}\langle X\rangle(\tau).\Theta$ and $\mathbb{A}\langle X\rangle(\tau).\mathbb{T}$, respectively. Further, the loop at lines 3-5 updates $\mathbb{T}$ on all $\theta' \in \{\theta\} \sqcup \Theta$; in case $\mathbb{T}$ was not defined on such a θ', then it will be defined after $e\langle\theta\rangle$ is processed. The definitional domain of $\mathbb{T}$ is thus continuously growing or potentially remains stationary as parametric events are processed, but it never decreases.

With these observations, we can prove *1.* easily by induction on τ. If $\tau = \epsilon$ then $Dom(\mathbb{A}\langle X\rangle(\epsilon).\mathbb{T}) = \mathbb{A}\langle X\rangle(\epsilon).\Theta = \Theta_\epsilon = \{\bot\}$. Suppose now that $Dom(\mathbb{A}\langle X\rangle(\tau).\mathbb{T}) = \mathbb{A}\langle X\rangle(\tau).\Theta = \Theta_\tau$ holds for $\tau \in \mathcal{E}\langle X\rangle^*$, and let $e\langle\theta\rangle \in \mathcal{E}\langle X\rangle$ be any parametric event. Then the following concludes the proof of *1.*:

$$
\begin{aligned}
& \mathit{Dom}(\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\mathbb{T}) \\
= \;& \mathit{Dom}(\mathbb{A}\langle X\rangle(\tau).\mathbb{T}) \cup (\{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta) \\
= \;& \mathbb{A}\langle X\rangle(\tau).\Theta \cup (\{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta) \\
= \;& (\{\bot\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta) \cup (\{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta) \\
= \;& \{\bot, \theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta \\
= \;& \mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\Theta \\
= \;& \{\bot, \theta\} \sqcup \Theta_\tau \\
= \;& \Theta_{\tau\, e\langle\theta\rangle}
\end{aligned}
$$

where the first equality follows from how the loop at lines 3-5 updates $\mathbb{T}$, the second by the induction hypothesis, the third by *5.* in Proposition 4, the fourth by *7.* in Proposition 4, the fifth by how Θ is updated at line 6, the sixth again by the induction hypothesis, and, finally, the seventh by Proposition 13.

Before we continue, let us first prove the following property:

> $\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\mathbb{T}(\theta') = \mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\max(\theta']_{\mathbb{A}\langle X\rangle(\tau).\Theta})\, e$ for any $e\langle\theta\rangle \in \mathcal{E}\langle X\rangle$ and any $\theta' \in \{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$.

One should be careful here to *not* get tricked thinking that this property is straightforward, because it says only what line 4 of $\mathbb{A}\langle X\rangle$ does. The complexity comes from the fact that if there were two different $\theta_1, \theta_2 \in \{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$ such that $\theta_1 = \max(\theta_2]_{\mathbb{A}\langle X\rangle(\tau).\Theta}$, then an unfortunate enumeration of the partial functions θ' in $\{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$ by the loop at lines 3-5 may lead to the non-parametric event e to be added twice to a slice: indeed, if θ_1 is processed before θ_2, then e is first added to the end of $\mathbb{T}(\theta_1)$ when $\theta' = \theta_1$, and then $\mathbb{T}(\theta_1)\, e$ is assigned to $\mathbb{T}(\theta_2)$ when $\theta' = \theta_2$; this way, $\mathbb{T}(\theta_2)$ ends up accumulating e twice instead of once, which is obviously wrong. Fortunately, since $\mathbb{A}\langle X\rangle(\tau).\Theta$ is lub closed (by *1.* above and Proposition 12), *3.* in Proposition 8 implies that there are no such different $\theta_1, \theta_2 \in \{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$. Therefore, there is no interference between the various assignments at line 4, regardless of the order in which the partial functions $\theta' \in \{\theta\} \sqcup \Theta$ are enumerated, which means that, indeed, $\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\mathbb{T}(\theta') = \mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\max(\theta']_{\mathbb{A}\langle X\rangle(\tau).\Theta})\, e$ for any $e\langle\theta\rangle \in \mathcal{E}\langle X\rangle$ and for any $\theta' \in \{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$. This lack of interference between updates of $\mathbb{T}$ also suggests an important implementation optimization: *the loop at lines 3-5 can be parallelized without having to duplicate the table $\mathbb{T}$!* Of course, the loop can be parallelized anyway if the table is duplicated and then merged within the original table, in the sense that all the writes to $\mathbb{T}(\theta')$ are done in a copy of $\mathbb{T}$. However, experiments show that the table $\mathbb{T}$ can be literally huge in real applications, in the order of billions of entries, so duplicating and merging it can be prohibitive.

2. can be now proved by induction on the length of τ. If $\tau = \epsilon$ then $\mathbb{A}\langle X\rangle(\epsilon).\Theta = \{\bot\}$, so $\theta' \in \mathbb{A}\langle X\rangle(\epsilon).\Theta$ can only be $\bot$; then $\mathbb{A}\langle X\rangle(\epsilon).\mathbb{T}(\bot) = \tau{\uparrow_\bot} = \epsilon$. Suppose now that $\mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\theta') = \tau{\uparrow_{\theta'}}$ for any $\theta' \in \mathbb{A}\langle X\rangle(\tau).\Theta$ and let us show that $\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\mathbb{T}(\theta') = (\tau\, e\langle\theta\rangle){\restriction_{\theta'}}$ for any $\theta' \in \mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\Theta$. As shown in the proof of *1.* above, $\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\Theta = \mathbb{A}\langle X\rangle(\tau).\Theta \cup (\{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta)$, so we have two cases to analyze. First, if $\theta' \in \{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$ then $\theta \sqsubseteq \theta'$ and so $(\tau\, e\langle\theta\rangle){\restriction_{\theta'}} = \tau{\uparrow_{\theta'}}\ e$; further,

$$\begin{aligned}
\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\mathbb{T}(\theta') &= \mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\max(\theta']_{\mathbb{A}\langle X\rangle(\tau).\Theta})\, e \\
&= \tau{\uparrow_{\max(\theta']_{\mathbb{A}\langle X\rangle(\tau).\Theta}}}\ e \\
&= \tau{\uparrow_{\theta'}}\ e \\
&= (\tau\, e\langle\theta\rangle){\restriction_{\theta'}},
\end{aligned}$$

where the first equality follows by the auxiliary property proved above, the second by the induction hypothesis using the fact that $\max(\theta']_{\mathbb{A}\langle X\rangle(\tau).\Theta} \in \mathbb{A}\langle X\rangle(\tau).\Theta$, and the third by Proposition 14.

Second, if $\theta' \in \mathbb{A}\langle X\rangle(\tau).\Theta$ but $\theta' \notin \{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$ then $\theta \not\sqsubseteq \theta'$ and so $(\tau\, e\langle\theta\rangle){\restriction_{\theta'}} = \tau{\uparrow_{\theta'}}$; further,

$$\begin{aligned}
\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\mathbb{T}(\theta') &= \mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\theta') \\
&= \tau{\uparrow_{\theta'}} \\
&= (\tau\, e\langle\theta\rangle){\restriction_{\theta'}},
\end{aligned}$$

where the first equality holds because θ' is not considered by the loop in lines 3-5 in $\mathbb{A}\langle X\rangle$, that is, $\theta' \notin \{\theta\} \sqcup \mathbb{A}\langle X\rangle(\tau).\Theta$, and the second equality follows by the induction hypothesis, as $\theta' \in \mathbb{A}\langle X\rangle(\tau).\Theta$. Therefore, $\mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\mathbb{T}(\theta') = (\tau\, e\langle\theta\rangle){\restriction_{\theta'}}$ for any $\theta' \in \mathbb{A}\langle X\rangle(\tau\, e\langle\theta\rangle).\Theta$, which completes the proof of *2.*

3. is the main result concerning our trace slicing algorithm and it follows now easily:

$$\begin{aligned}
\tau{\uparrow_\theta} &= \tau{\uparrow_{\max(\theta]_{\Theta_\tau}}} \\
&= \tau{\uparrow_{\max(\theta]_{\mathbb{A}\langle X\rangle(\tau).\Theta}}} \\
&= \mathbb{A}\langle X\rangle(\tau).\mathbb{T}(\max(\theta]_{\mathbb{A}\langle X\rangle(\tau).\Theta})
\end{aligned}$$

The first equality follows by Proposition 14, the second by *1.* and the third by *2.*, as $\max(\theta]_{\mathbb{A}\langle X\rangle(\tau).\Theta} \in \mathbb{A}\langle X\rangle(\tau).\Theta$. This concludes the correctness proof of our trace slicing algorithm $\mathbb{A}\langle X\rangle$. □

4.3 Parametric Monitoring

We next formalize the notion of parametric monitors and introduces two algorithms for parametric monitoring.

4.3.1 Parametric Monitors

In this section we first define monitors M as a variant of Moore machines with potentially infinitely many states; then we define parametric monitors $\Lambda X.M$ as monitors maintaining one state of M per parameter instance. Like for parametric properties, which turned out to be just properties over parametric traces, we show that parametric monitors are also just monitors, but for parametric events and with instance-indexed states and output categories. We also show that a parametric monitor $\Lambda X.M$ is a monitor for the parametric property $\Lambda X.P$, with P the property monitored by M.

The Non-Parametric Case

We start by defining non-parametric monitors as a variant of Moore machines [69] that allows infinitely many states:

Definition 18 *A **monitor** M is a tuple $(S, \mathcal{E}, \mathcal{C}, i, \sigma : S \times \mathcal{E} \to S, \gamma : S \to \mathcal{C})$, where S is a set of states, $\mathcal{E}$ is a set of input events, $\mathcal{C}$ is a set of output categories, $i \in S$ is the initial state, σ is the transition function, and γ is the output function. The transition function is extended to $\sigma : S \times \mathcal{E}^* \to S$ as expected: $\sigma(s, \epsilon) = s$ and $\sigma(s, we) = \sigma(\sigma(s, w), e)$ for any $s \in S$, $e \in \mathcal{E}$, and $w \in \mathcal{E}^*$.*

The notion of a monitor above is rather conceptual. Actual implementations of monitors need not generate all the state space apriori, but on a "by need" basis. Consider, for example, a monitor for a property specified using an NFA: the monitor performs an NFA-to-DFA construction on the fly, as events are received, thus generating only those states in the DFA that are needed by the monitored execution trace; since generation of next set of states is fast, one need not even hash the generated DFA states, the entire memory needed by monitor staying linear in the size of the NFA.

Allowing monitors with infinitely many states is a necessity in our context. Even though only a finite number of states is reached during any given (finite) execution trace, there is, in general, no bound on how many states are reached. For example, monitors for context-free grammars like the ones in [67] have potentially unbounded stacks as part of their state. Also, as shown shortly, parametric monitors have domains of functions as state spaces, which are infinite as well. What is common to all monitors though is that they can take a trace event-by-event and, as each event is processed, classify the observed trace into a category. The following is natural:

Definition 19 *$M = (S, \mathcal{E}, \mathcal{C}, i, \sigma, \gamma)$ is a **monitor for property** $P : \mathcal{E}^* \to \mathcal{C}$ iff $\gamma(\sigma(i, w)) = P(w)$ for each $w \in \mathcal{E}^*$.*

Since we allow monitors to have infinitely many states, there is a strong correspondence between properties and monitors:

Proposition 15 *Every monitor M defines a property $\mathcal{P}_M$ with M a monitor for $\mathcal{P}_M$. Every property P defines a monitor $\mathcal{M}_P$ with $\mathcal{M}_P$ a monitor for P. For any property P, $\mathcal{P}_{\mathcal{M}_P} = P$.*

Proof: Given $M = (S, \mathcal{E}, \mathcal{C}, i, \sigma, \gamma)$, let $\mathcal{P}_M : \mathcal{E}^* \to \mathcal{C}$ be the property $\mathcal{P}_M(w) = \gamma(\sigma(i, w))$; note that M is indeed a monitor for $\mathcal{P}_M$. Given $P : \mathcal{E}^* \to \mathcal{C}$, let $\mathcal{M}_P$ be the monitor $(S_P, \mathcal{E}, \mathcal{C}, i_P, \sigma_P, \gamma_P)$ with $S_P = \mathcal{E}^*$, $i_P = \epsilon$, $\sigma_P(w, e) = we$, $\gamma_P(w) = P(w)$. $\mathcal{M}_P$ is a monitor for P as $\gamma_P(\sigma_P(i_P, w)) = \gamma_P(\sigma_P(\epsilon, w)) = \gamma_P(\epsilon w) = \gamma_P(w) = P(w)$. Finally, $\mathcal{P}_{\mathcal{M}_P}(w) = \gamma_P(\sigma_P(i_P, w)) = P(w)$ for any $w \in \mathcal{E}^*$, so $\mathcal{P}_{\mathcal{M}_P} = P$. □

The equality of monitors $\mathcal{M}_{\mathcal{P}_M} = M$ does not hold for any monitor M; it does hold when $M = \mathcal{M}_P$ for some property P.

Definition 20 *Monitors M and M' are **property equivalent**, or just **equivalent**, written $M \equiv M'$, iff they are monitors for the same property; with the notation above, $M \equiv M'$ iff $\mathcal{P}_M = \mathcal{P}_{M'}$.*

Corollary 4 *With the notation in Proposition 15, $\mathcal{M}_{\mathcal{P}_M} \equiv M$.*

The Parametric Case

We next define parametric monitors in the same style as the other parametric entities defined in this paper: starting with a base monitor and a set of parameters, the corresponding parametric monitor can be thought of as a set of base monitors running in parallel, one for each parameter instance.

Definition 21 *Given parameters X with corresponding values V_X and monitor $M = (S, \mathcal{E}, \mathcal{C}, i, \sigma : S \times \mathcal{E} \to S, \gamma : S \to \mathcal{C})$, we define the **parametric monitor** $\Lambda X.M$ as the monitor*

$$([[X \rightharpoonup V] \to S], \mathcal{E}\langle X \rangle, [[X \rightharpoonup V] \to \mathcal{C}], \lambda\theta.i, \Lambda X.\sigma, \Lambda X.\gamma),$$

with $\Lambda X.\sigma : [[X \rightharpoonup V] \to S] \times \mathcal{E}\langle X \rangle \to [[X \rightharpoonup V] \to S]$ and $\Lambda X.\gamma : [[X \rightharpoonup V] \to S] \to [[X \rightharpoonup V] \to \mathcal{C}]$ defined

as

$$(\Lambda X.\sigma)(\delta, e\langle\theta'\rangle)(\theta) = \begin{cases} \sigma(\delta(\theta), e) & \text{if } \theta' \sqsubseteq \theta \\ \delta(\theta) & \text{if } \theta' \not\sqsubseteq \theta \end{cases}$$

$$(\Lambda X.\gamma)(\delta)(\theta) = \gamma(\delta(\theta))$$

for any $\delta \in [[X \rightarrow V] \rightarrow S]$ *and any* $\theta, \theta' \in [X \rightarrow V]$.

Therefore, a state δ of parametric monitor $\Lambda X.M$ maintains a state $\delta(\theta)$ of M for each parameter instance θ, takes parametric events as input, and outputs categories indexed by parameter instances (one output category of M per parameter instance).

Proposition 16 *If M is a monitor for property P then parametric monitor $\Lambda X.M$ is a monitor for parametric property $\Lambda X.P$, or, with the notation in Proposition 15, $\mathcal{P}_{\Lambda X.M} = \Lambda X.\mathcal{P}_M$.*

Proof: We show that $(\Lambda X.\gamma)((\Lambda X.\sigma)(\lambda\theta.i, \tau)) = (\Lambda X.P)(\tau)$ for any $\tau \in \mathcal{E}\langle X\rangle^*$, i.e., after application on $\theta \in [X \rightarrow V]$, that $\gamma((\Lambda X.\sigma)(\lambda\theta.i, \tau)(\theta)) = P(\tau{\restriction_\theta})$ for any $\tau \in \mathcal{E}\langle X\rangle^*$ and $\theta[X \rightarrow V]$. Since M is a monitor for P, it suffices to show that $(\Lambda X.\sigma)(\lambda\theta.i, \tau)(\theta) = \sigma(i, \tau{\restriction_\theta})$ for any $\tau \in \mathcal{E}\langle X\rangle^*$ and $\theta[X \rightarrow V]$. We prove it by induction on τ. If $\tau = \epsilon$ then $(\Lambda X.\sigma)(\lambda\theta.i, \epsilon)(\theta) = (\lambda\theta.i)(\theta) = i = \sigma(i, \epsilon) = \sigma(i, \epsilon{\restriction_\theta})$. Suppose now that $(\Lambda X.\sigma)(\lambda\theta.i, \tau)(\theta) = \sigma(i, \tau{\restriction_\theta})$ for some arbitrary but fixed $\tau \in \mathcal{E}\langle X\rangle^*$ and for any $\theta \in [X \rightarrow V]$, and let $e\langle\theta'\rangle$ be any parametric event in $\mathcal{E}\langle X\rangle$ and let $\theta \in [X \rightarrow V]$ be any parameter instance. The inductive step is then as follows:

$$\begin{aligned}
(\Lambda X.\sigma)(\lambda\theta.i, \tau\, e\langle\theta'\rangle)(\theta) &= (\Lambda X.\sigma)((\Lambda X.\sigma)(\lambda\theta.i, \tau), e\langle\theta'\rangle)(\theta) \\
&= (\Lambda X.\sigma)(\sigma(i, \tau{\restriction_\theta}), e\langle\theta'\rangle)(\theta) \\
&= \begin{cases} \sigma(\sigma(i, \tau{\restriction_\theta}), e) & \text{if } \theta' \sqsubseteq \theta \\ \sigma(i, \tau{\restriction_\theta}) & \text{if } \theta' \not\sqsubseteq \theta \end{cases} \\
&= \begin{cases} \sigma(i, \tau{\restriction_\theta}\ e) & \text{if } \theta' \sqsubseteq \theta \\ \sigma(i, \tau{\restriction_\theta}) & \text{if } \theta' \not\sqsubseteq \theta \end{cases} \\
&= \sigma(i, (\tau\, e\langle\theta'\rangle){\restriction_\theta})
\end{aligned}$$

The first equality above follows by the second part of Definition 18), the second by the induction hypothesis, the third by Definition 21, the fourth again by the second part of Definition 18, and the fifth by Definition 7. This concludes our proof. □

```
Algorithm 𝔹⟨X⟩(M = (S, ℰ, 𝒞, i, σ, γ))
Input: finite parametric trace τ ∈ ℰ⟨X⟩*
Output: mapping Γ : [[X→V]→𝒞] and set Θ ⊆ [X→V]

1  Δ ← ⊥; Δ(⊥) ← i; Θ ← {⊥}
2  for all parametric event e⟨θ⟩ in order in τ do
3  ⋮ for all θ′ ∈ {θ} ⊔ Θ do
4  ⋮ ⋮ Δ(θ′) ← σ(Δ(max(θ′]_Θ), e)
5  ⋮ ⋮ Γ(θ′) ← γ(Δ(θ′))        // a message may be output here
6  ⋮ endfor
7  ⋮ Θ ← {⊥, θ} ⊔ Θ
8  endfor
```

Figure 4.5: Parametric monitoring algorithm $\mathbb{B}\langle X\rangle$

4.3.2 General Parametric Monitoring Algorithm $\mathbb{B}\langle X\rangle$

We next propose a monitoring algorithm for parametric properties. Analyzing the definition of a parametric monitor (Definition 21), the first thing we note is that its state space is not only infinite, but it is not even enumerable. Therefore, a first challenge in monitoring parametric properties is how to represent the states of the parametric monitor. Inspired by the algorithm for trace slicing in Figure 4.4, we encode functions $[[X\rightarrow V]\rightarrow S]$ as tables with entries indexed by parameter instances in $[X\rightarrow V]$ and with contents states in S. Following similar arguments as in the proof of the trace slicing algorithm, such tables will have a finite number of entries provided that each event instantiates only a finite number of parameters.

Figure 4.5 shows our monitoring algorithm for parametric properties. Given parametric property $\Lambda X.P$ and M a monitor for P, $\mathbb{B}\langle X\rangle(M)$ yields a monitor that is equivalent to $\Lambda X.M$, that is, a monitor for $\Lambda X.P$. Section 4.3.3 shows one way to use this algorithm: a monitor M is first synthesized from the base property P, then that monitor M is used to synthesize the monitor $\mathbb{B}\langle X\rangle(M)$ for the parametric property $\Lambda X.P$. $\mathbb{B}\langle X\rangle(M)$ follows very closely the algorithm for trace slicing in Figure 4.4, the main difference being that trace slices are processed, as generated, by M: instead of calculating the trace slice of θ' by appending base event e to the corresponding existing trace slice in line 4 of $\mathbb{A}\langle X\rangle$, we now calculate and store in table Δ the state of the "monitor instance" corresponding to θ' by sending e to the corresponding existing monitor instance (line 4 in $\mathbb{B}\langle X\rangle(M)$); at the same time we also calculate the output corresponding to that monitor instance and store it in table Γ. In other words, we replace trace slices in $\mathbb{A}\langle X\rangle$ by local monitors processing online those slices. In our implementation in Section 4.3.3, we also check whether $\Gamma(\theta')$ at line 5 violates the property and, if so, an error message including θ' is output to the user.

Definition 22 *Given $\tau \in \mathcal{E}\langle X\rangle^*$, let $\mathbb{B}\langle X\rangle(M)(\tau).\Theta$ and $\mathbb{B}\langle X\rangle(M)(\tau).\Delta$ and $\mathbb{B}\langle X\rangle(M)(\theta).\Gamma$ be the three data-structures maintained by the algorithm $\mathbb{B}\langle X\rangle(M)$ in Figure 4.5 after processing τ. Let $\bot \mapsto i = \mathbb{B}\langle X\rangle(M)(\epsilon).\Delta \in [[X \rightharpoonup V] \rightharpoonup S]$ be the partial map taking $\bot \in [X \rightharpoonup V]$ to i and undefined elsewhere.*

Corollary 5 *The following hold for any $\tau \in \mathcal{E}\langle X\rangle^*$:*

1. *$Dom(\mathbb{B}\langle X\rangle(M)(\tau).\Delta) = \mathbb{B}\langle X\rangle(M)(\tau).\Theta = \Theta_\tau$;*
2. *$\mathbb{B}\langle X\rangle(M)(\tau).\Delta(\theta) = \sigma(i, \tau\restriction_\theta)$ and*
   $\mathbb{B}\langle X\rangle(M)(\tau).\Gamma(\theta) = \gamma(\sigma(i, \tau\restriction_\theta))$
   for any $\theta \in \mathbb{B}\langle X\rangle(M)(\tau).\Theta$;
3. *$\sigma(i, \tau\restriction_\theta) = \mathbb{B}\langle X\rangle(M)(\tau).\Delta(\max(\theta]_{\mathbb{B}\langle X\rangle(M)(\tau).\Theta})$ and*
   $\gamma(\sigma(i, \tau\restriction_\theta)) = \mathbb{B}\langle X\rangle(M)(\tau).\Gamma(\max(\theta]_{\mathbb{B}\langle X\rangle(M)(\tau).\Theta})$
   for any $\theta \in [X \rightharpoonup V]$.

Proof: Follows from Theorem 1 and the discussion above. □

We next associate a monitor to the algorithm in Figure 4.5:

Definition 23 *For the algorithm $\mathbb{B}\langle X\rangle(M)$ in Figure 4.5, let $\mathcal{M}_{\mathbb{B}\langle X\rangle(M)} = (R, \mathcal{E}\langle X\rangle, [[X \rightharpoonup V] \to \mathcal{C}], , \bot \mapsto i, next, out)$ be the monitor defined as follows: $R \subseteq [[X \rightharpoonup V] \rightharpoonup S]$ is the set $\{\mathbb{B}\langle X\rangle(M)(\tau).\Delta \mid \tau \in \mathcal{E}\langle X\rangle^*\}$ of reachable Δ's in $\mathbb{B}\langle X\rangle(M)$, and $next : R \times \mathcal{E}\langle X\rangle \to R$ and $out : R \to [[X \rightharpoonup V] \to \mathcal{C}]$ are the functions defined as follows ($\tau \in \mathcal{E}\langle X\rangle^*$, $e \in \mathcal{E}$, $\theta \in [X \rightharpoonup V]$):*

$$next(\mathbb{B}\langle X\rangle(M)(\tau).\Delta, e\langle\theta\rangle) = \mathbb{B}\langle X\rangle(M)(\tau\, e\langle\theta\rangle).\Delta, \text{ and}$$
$$out(\mathbb{B}\langle X\rangle(M)(\tau).\Delta)(\theta) = \mathbb{B}\langle X\rangle(M)(\tau).\Gamma(\max(\theta]_{\mathbb{B}\langle X\rangle(M)(\tau).\Theta}).$$

Theorem 2 *$\mathcal{M}_{\mathbb{B}\langle X\rangle(M)} \equiv \Lambda X.M$ for any monitor M.*

Proof: We have to show that, for any $\tau \in \mathcal{E}\langle X\rangle^*$, $out(next(\bot \mapsto i, \tau))$ and $(\Lambda X.\gamma)((\Lambda X.\sigma)(\lambda\theta.i, \tau))$ are equal as total functions in $[[X \rightharpoonup V] \to \mathcal{C}]$. Let $\theta \in [X \rightharpoonup V]$; then:

$$\begin{aligned}
out(next(\bot \mapsto i, \tau))(\theta) &= out(\mathbb{B}\langle X\rangle(M)(\tau).\Delta)(\theta) \\
&= \mathbb{B}\langle X\rangle(M)(\tau).\Gamma(\max(\theta]_{\mathbb{B}\langle X\rangle(M)(\tau).\Theta}) \\
&= \gamma(\sigma(\lambda\theta.i, \tau\restriction_\theta)) \\
&= \gamma((\Lambda X.\sigma)(\lambda\theta.i, \tau)(\theta)) \\
&= (\Lambda X.\gamma)((\Lambda X.\sigma)(\lambda\theta.i, \tau))(\theta).
\end{aligned}$$

The first equality above follows inductively by the definition of *next* (Definition 23), noticing that $\bot \mapsto i = \mathbb{B}\langle X\rangle(M)(\epsilon).\Delta$. The second equality follows by the definition of *out* (Definition 23) and the third by *3.* in Corollary 5. The fourth equality above follows inductively by the definition of $\Lambda X.\sigma$ (Definition 21) and has already been proved as part of the proof of Proposition 16. Finally, the fifth equality follows by the definition of $\Lambda X.\gamma$ (Definition 21).

Therefore, $\mathcal{M}_{\mathbb{B}\langle X\rangle(M)}$ and $\Lambda X.M$ define the same property. □

Corollary 6 *If M is a monitor for P and X is a set of parameters, then $\mathcal{M}_{\mathbb{B}\langle X\rangle(M)}$ is a monitor for parametric property $\Lambda X.P$.*

Proof: With the notation in Proposition 15, Theorem 2 implies that $\mathcal{P}_{\mathcal{M}_{\mathbb{B}\langle X\rangle(M)}} = \mathcal{P}_{\Lambda X.M}$. By Proposition 16 and the fact that $P = \mathcal{P}_M$, we conclude that $\mathcal{P}_{\mathcal{M}_{\mathbb{B}\langle X\rangle(M)}} = \Lambda X.P$. □

4.3.3 Online Parametric Monitoring Algorithm $\mathbb{C}\langle X\rangle$

Algorithm $\mathbb{C}\langle X\rangle$ in Figure 4.6 refines Algorithm $\mathbb{B}\langle X\rangle$ in Figure 4.5 for efficient online monitoring. Since no complete trace is given in online monitoring, $\mathbb{C}\langle X\rangle$ focuses on actions to carry out when a parametric event $e\langle\theta\rangle$ arrives; in other words, it essentially expands the body of the outer loop in $\mathbb{B}\langle X\rangle$ (lines 3 to 7 in Figure 4.5). We chose not to use $\mathbb{B}\langle X\rangle$ directly for our implementation for efficiency concerns: the inner loop in $\mathbb{B}\langle X\rangle$ requires search for all parameter instances in Θ that are compatible with θ; this search can be very expensive. $\mathbb{C}\langle X\rangle$ introduces an auxiliary data structure and illustrates a mechanical way to accomplish the search, which also facilitates optimizations to improve the performance of monitoring. While $\mathbb{B}\langle X\rangle$ did not require that θ in $e\langle\theta\rangle$ be of finite domain, $\mathbb{C}\langle X\rangle$ needs that requirement to terminate. Note that in practice $Dom(\theta)$ is always finite (because the program state is finite).

$\mathbb{C}\langle X\rangle$ uses three tables: $\Delta, \mathcal{U}$ and Γ. Δ and Γ are the same as Δ and Γ in $\mathbb{B}\langle X\rangle$, respectively. $\mathcal{U}$ is an auxiliary data structure used to optimize the search "for all $\theta' \in \{\theta\} \sqcup \Theta$" in $\mathbb{B}\langle X\rangle$ (line 3 in Figure 4.5). It maps each parameter instance θ into the finite set of parameter instances encountered in Δ so far that are strictly more informative than θ, i.e., $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in Dom(\Delta) \text{ and } \theta \sqsubset \theta'\}$. Another major difference between $\mathbb{B}\langle X\rangle$ and $\mathbb{C}\langle X\rangle$ is that $\mathbb{C}\langle X\rangle$ does *not* maintain Θ during computation; instead, Θ is implicitly captured by the domain of Δ in $\mathbb{C}\langle X\rangle$. Intuitively, Θ at the beginning/end of the body of the outer loop in $\mathbb{B}\langle X\rangle$ is $Dom(\Delta)$ at the beginning/end of $\mathbb{C}\langle X\rangle$, respectively. However, Θ is fixed during the loop at lines 3 to 6 in $\mathbb{B}\langle X\rangle$ and updated atomically in line 7, while $Dom(\Delta)$ can be changed at any time during the execution of $\mathbb{C}\langle X\rangle$.

```
Algorithm ℂ⟨X⟩(M = (S, ℰ, 𝒞, i, σ, γ))
Globals: mapping Δ : [[X→V]→S] and
         mapping 𝒰 : [X→V] → 𝒫_f([X→V]) and
         mapping Γ : [[X→V]→𝒞]
Initialization: 𝒰(θ) ← ∅ for any θ ∈ [X→V], Δ(⊥) ← i

function main(e⟨θ⟩)
 1  if Δ(θ)undefinedthen
 2  ⋮ for all θ_max ⊏ θ (in reversed topological order) do
 3  ⋮ ⋮ if Δ(θ_max) definedthen
 4  ⋮ ⋮ ⋮ goto 7
 5  ⋮ ⋮ endif
 6  ⋮ endfor
 7  ⋮ defineTo(θ, θ_max)
 8  ⋮ for all θ_max ⊏ θ (in reversed topological order) do
 9  ⋮ ⋮ for all θ_comp ∈ 𝒰(θ_max) that is compatible with θ do
10  ⋮ ⋮ ⋮ if Δ(θ_comp ⊔ θ) undefined then
11  ⋮ ⋮ ⋮ ⋮ defineTo(θ_comp ⊔ θ, θ_comp)
12  ⋮ ⋮ ⋮ endif
13  ⋮ ⋮ endfor
14  ⋮ endfor
15  endif
16  for all θ' ∈ {θ} ∪ 𝒰(θ) do
17  ⋮ Δ(θ') ← σ(Δ(θ'), e)
18  ⋮ Γ(θ') ← σ(Δ(θ'))
19  endfor

function defineTo(θ, θ')
1  Δ(θ) ← Δ(θ')
2  for all θ'' ⊏ θ do
3  ⋮ 𝒰(θ'') ← 𝒰(θ'') ∪ {θ}
4  endfor
```

Figure 4.6: Online parametric monitoring algorithm $\mathbb{C}\langle X\rangle$

$\mathbb{C}\langle X\rangle$ is composed of two functions, main and defineTo. The defineTo function takes two parameter instances, θ and θ', and adds a new entry corresponding to θ into Δ and $\mathcal{U}$. Specifically, it sets $\Delta(\theta)$ to $\Delta(\theta')$ and adds θ into the set $\mathcal{U}(\theta'')$ for each $\theta'' \sqsubset \theta$.

The main function differentiates two cases when a new event $e\langle\theta\rangle$ is received and processed. The simpler case is that Δ is already defined on θ, i.e., $\theta \in \Theta$ at the beginning of the iteration of the outer loop in $\mathbb{B}\langle X\rangle$. In this case, $\{\theta\} \sqcup \Theta = \{\theta' \mid \theta' \in \Theta \text{ and } \theta \sqsubseteq \theta'\} \subseteq \Theta$, so the lines 3 to 6 in $\mathbb{B}\langle X\rangle$ become precisely the lines 16 to 19 in $\mathbb{C}\langle X\rangle$. In the other case, when Δ is not already defined on θ, main takes two steps to handle e. The first step searches for new parameter instances introduced by $\{\theta\} \sqcup \Theta$ and adds entries for them into Δ (lines 2 to 15). We first add an entry to Δ for θ at lines 2 to 7. Then we search for all parameter instances θ_{comp} that are compatible with θ, making

use of $\mathcal{U}$ (line 8 and 9); for each such θ_{comp}, an appropriate entry is added to Δ for its lub with θ, and $\mathcal{U}$ updated accordingly (lines 10 to 12). This way, Δ will be defined on all the new parameter instances introduced by $\{\theta\} \sqcup \Theta$ after the first step. In the second step, the related monitor states and outputs are updated in a similar way as in the first case (lines 16 to 19). It is interesting to note how $\mathbb{C}\langle X\rangle$ searches at lines 2 and 8 for the parameter instance $\max(\theta]_\Theta$ that $\mathbb{B}\langle X\rangle$ refers to at line 4 in Figure 4.5: it enumerates all the $\theta_{max} \sqsubset \theta$ in reversed topological order (from larger to smaller); *1.* in Proposition 8 guarantees that the maximum exists and, since it is unique, our search will find it.

Correctness of $\mathbb{C}\langle X\rangle$**.** We next prove the correctness of $\mathbb{C}\langle X\rangle$ by showing that it is equivalent to the body of the outer loop in $\mathbb{B}\langle X\rangle$. Suppose that parametric trace τ has already been processed by both $\mathbb{C}\langle X\rangle$ and $\mathbb{B}\langle X\rangle$, and a new event $e\langle\theta\rangle$ is to be processed next.

Let us first note that $\mathbb{C}\langle X\rangle$ terminates if $Dom(\theta)$ is finite. Indeed, then there is only a finite number of partial maps less informative than θ, that is, only a finite number of iterations for the loops at lines 2 and 8 in main; since $\mathcal{U}$ is only updated at line 3 in defineTo, $\mathcal{U}(\theta)$ is finite for any $\theta \in [X \rightharpoonup V]$ and thus the loop at line 9 in main also terminates. Assuming that running the base monitor M takes constant time, the worse case complexity of $\mathbb{C}\langle X\rangle(M)$ is $O(n \times m)$ to process $e\langle\theta\rangle$, where n is $2^{|Dom(\theta)|}$ and m is the number of incompatible parameter instances in τ. Parametric properties often have a fixed and small number of parameters, in which case n is not significant. Depending on the trace, m can unavoidably grow arbitrarily large; in the worst case, each event may carry an instance incompatible with the previous ones.

Lemma 1 *$\mathcal{U}(\theta) = \{\theta' \mid \theta' \in Dom(\Delta)$ and $\theta \sqsubset \theta'\}$ before and after each execution of* defineTo*, for all $\theta \in [X \rightharpoonup V]$.*

Proof: By how $\mathbb{C}\langle X\rangle$ is initialized, for any $\theta \in [X \rightharpoonup V]$ we have $\emptyset = \mathcal{U}(\theta) = \{\theta' \mid \theta' \in Dom(\Delta)$ and $\theta \sqsubset \theta'\}$ before the first execution of defineTo. Now suppose that $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in Dom(\Delta)$ and $\theta \sqsubset \theta'\}$ for any $\theta \in [X \rightharpoonup V]$ before an execution of defineTo and show that it also holds after the execution of defineTo. Since defineTo(θ, θ') adds a new parameter instance θ into $Dom(\Delta)$ and also adds θ into the set $\mathcal{U}(\theta'')$ for any $\theta'' \in [X \rightharpoonup V]$ with $\theta'' \sqsubset \theta$, we still have $\mathcal{U}(\theta) = \{\theta' \mid \theta' \in Dom(\Delta)$ and $\theta \sqsubset \theta'\}$ for any $\theta \in [X \rightharpoonup V]$ after the execution of defineTo. Also, the only way $\mathbb{C}\langle X\rangle$ adds a new parameter instance θ into $Dom(\Delta)$ is using defineTo. Therefore the lemma holds. □

Next result establishes the correctness of our implementation. We use the following notation. Recall that we fixed parametric trace τ and event $e\langle\theta\rangle$. Let $\mathcal{U}_\mathbb{C}$, $\Delta_\mathbb{C}$, and $\Gamma_\mathbb{C}$ be the three data-

structures maintained by $\mathbb{C}\langle X\rangle(M)$ for some M. Let $\Delta^b_{\mathbb{C}}$ and $\Gamma^b_{\mathbb{C}}$ be the $\Delta_{\mathbb{C}}$ and $\Gamma_{\mathbb{C}}$ when $\mathsf{main}(e\langle\theta\rangle)$ begins ("b" stays for "at the beginning"); let $\Delta^e_{\mathbb{C}}$ and $\Gamma^e_{\mathbb{C}}$ be the $\Delta_{\mathbb{C}}$ and $\Gamma_{\mathbb{C}}$ when $\mathsf{main}(e\langle\theta\rangle)$ ends ("e" stays for "at the end"; and let $\Delta^m_{\mathbb{C}}$ and $\mathcal{U}^m_{\mathbb{C}}$ be the $\Delta_{\mathbb{C}}$ and $\mathcal{U}_{\mathbb{C}}$ when $\mathsf{main}(e\langle\theta\rangle)$ reaches line 16 ("m stays for "in the middle").

Theorem 3 *The following hold:*

1. $Dom(\Delta^m_{\mathbb{C}}) = \{\bot, \theta\} \sqcup Dom(\Delta^b_{\mathbb{C}})$;
2. $\Delta^m_{\mathbb{C}}(\theta') = \Delta^m_{\mathbb{C}}(\max(\theta']_{Dom(\Delta^b_{\mathbb{C}})})$, *for all* $\theta' \in Dom(\Delta^m_{\mathbb{C}})$;
3. *If* $\Delta^b_{\mathbb{C}} = \mathbb{B}\langle X\rangle(M)(\tau).\Delta$ *and* $\Gamma^b_{\mathbb{C}} = \mathbb{B}\langle X\rangle(M)(\tau).\Gamma$, *then* $\Delta^e_{\mathbb{C}} = \mathbb{B}\langle X\rangle(M)(\tau\, e\langle\theta\rangle).\Delta$ *and* $\Gamma^b_{\mathbb{C}} = \mathbb{B}\langle X\rangle(M)(\tau\, e\langle\theta\rangle).\Gamma$.

Proof: Let $\Theta_{\mathbb{C}} = Dom(\Delta^b_{\mathbb{C}}) = Dom(\Delta_{\mathbb{B}}(\tau))$ and $\Delta_{\mathbb{B}}(\tau) = \mathbb{B}\langle X\rangle(M)(\tau\langle\theta\rangle).\Delta$ for simplicity.

1. There are two cases to analyze, depending upon θ is in $\Theta_{\mathbb{C}}$ or not. If $\theta \in \Theta_{\mathbb{C}}$ then lines 2 to 14 are skipped and $Dom(\Delta_{\mathbb{C}})$ remains unchanged, that is, $\{\bot, \theta\} \sqcup \Theta_{\mathbb{C}} = \Theta_{\mathbb{C}} = Dom(\Delta^b_{\mathbb{C}}) = Dom(\Delta^m_{\mathbb{C}})$ when $\mathsf{main}(e\langle\theta\rangle)$ reaches line 16. If $\theta \notin \Theta_{\mathbb{C}}$ then lines 2 to 14 are executed to add new parameter instances into $Dom(\Delta_{\mathbb{C}})$. First, an entry for θ will be added to $\Delta_{\mathbb{C}}$ at line 7. Second, an entry for $\theta_{comp} \sqcup \theta$ will be added to $\Delta_{\mathbb{C}}$ at line 11 (if $\Delta_{\mathbb{C}}$ not already defined on $\theta_{comp} \sqcup \theta$) eventually for any $\theta_{comp} \in \Theta_{\mathbb{C}}$ compatible with θ: that is because θ_{max} can also be $\bot$ at line 8, in which case Lemma 1 implies that $\mathcal{U}(\theta_{max}) = \Theta_{\mathbb{C}}$. Therefore, when line 16 is reached, $Dom(\Delta^m_{\mathbb{C}})$ is defined on all the parameter instances in $\{\theta\} \cup (\{\theta\} \sqcup \Theta_{\mathbb{C}})$. Since $\bot \in \Theta_{\mathbb{C}}$, the latter equals $\{\theta\} \sqcup \Theta_{\mathbb{C}}$, and since $\Delta^m_{\mathbb{C}}$ remains defined on $\Theta_{\mathbb{C}}$, we conclude that $\Delta^m_{\mathbb{C}}$ is defined on all instances in $(\{\theta\} \sqcup \Theta_{\mathbb{C}}) \cup \Theta_{\mathbb{C}}$, which by *5.* and *7.* in Proposition 4 equals $\{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}$.

2. We analyze the same two cases as above. If $\theta \in \Theta_{\mathbb{C}}$ then lines 2 to 14 are skipped and $Dom(\Delta_{\mathbb{C}})$ remains unchanged. Then $\max(\theta']_{\Theta_{\mathbb{C}}} = \theta'$ for each $\theta' \in Dom(\Delta^m_{\mathbb{C}})$, so the result follows. Suppose now that $\theta \notin \Theta_{\mathbb{C}}$. By *1.* and its proof, each $\theta' \in Dom(\Delta^m_{\mathbb{C}})$ is either in $\Theta_{\mathbb{C}}$ or otherwise in $(\{\theta\} \sqcup \Theta_{\mathbb{C}}) - \Theta_{\mathbb{C}}$. The result immediately holds when $\theta' \in \Theta_{\mathbb{C}}$ as $\max(\theta']_{\Theta_{\mathbb{C}}} = \theta'$ and $\Delta(\theta')$ stays unchanged until line 16. If $\theta' \in (\{\theta\} \sqcup \Theta_{\mathbb{C}}) - \Theta_{\mathbb{C}}$ then $\Delta(\theta')$ is set at either line 7 ($\theta' = \theta$) or at line 11 ($\theta' \neq \theta$):

(a) For line 7, the loop at lines 2 to 6 checks all the parameter instances that are less informative than θ to find the first one in $\Theta_{\mathbb{C}}$ in reversed topological order (i.e., if $\theta_1 \sqsubset \theta_2$ then θ_2 will be checked before θ_1). Since by *1.* in Proposition 8 we know that $\max(\theta]_{\Theta_{\mathbb{C}}} \in \Theta_{\mathbb{C}}$ exists (and it is unique), the loop at lines 2 to 6 will break precisely when $\theta_{max} = \max(\theta]_{\Theta_{\mathbb{C}}}$, so the result holds when $\theta' = \theta$

because of the entry introduced for θ in $\Delta_{\mathbb{C}}$ at line 7 and because the remaining lines 8 to 14 do not change $\Delta_{\mathbb{C}}(\theta)$.

(b) When $\Delta_{\mathbb{C}}(\theta')$ is set at line 11, note that the loop at lines 8 to 14 also iterates over all $\theta_{max} \sqsubset \theta$ in reversed topological order, so $\theta' = \theta_{comp} \sqcup \theta$ for some $\theta_{comp} \in \Theta_{\mathbb{C}}$ compatible with θ such that $\theta_{max} \sqsubset \theta_{comp}$, where $\theta_{max} \sqsubset \theta$ is such that there is no other θ'_{max} with $\theta_{max} \sqsubset \theta'_{max} \sqsubset \theta$ and $\theta' = \theta'_{comp} \sqcup \theta$ for some $\theta'_{comp} \in \Theta_{\mathbb{C}}$ compatible with θ such that $\theta'_{max} \sqsubset \theta'_{comp}$. We claim that there is only one such θ_{comp}, which is precisely $\max(\theta']_{\Theta_{\mathbb{C}}}$: Let θ'_{comp} be the parameter instance $\max(\theta']_{\Theta_{\mathbb{C}}}$. The above implies that $\theta_{comp} \sqsubseteq \theta'_{comp} \sqsubseteq \theta'$. Also, $\theta'_{comp} \sqcup \theta = \theta'$ because $\theta' = \theta_{comp} \sqcup \theta \sqsubseteq \theta'_{comp} \sqcup \theta \sqsubseteq \theta'$. Let θ'_{max} be $\theta'_{comp} \sqcap \theta$, that is, the largest with $\theta'_{max} \sqsubseteq \theta'_{comp}$ and $\theta'_{max} \sqsubseteq \theta$ (we let its existence as exercise). It is relatively easy to see now that $\theta_{comp} \sqsubset \theta'_{comp}$ implies $\theta_{max} \sqsubset \theta'_{max}$ (we let it as an exercise, too), which contradicts the assumption of this case that $\Delta_{\mathbb{C}}$ was not defined on θ'. Therefore, $\theta_{comp} = \max(\theta']_{\Theta_{\mathbb{C}}}$ before line 11 is executed, which means that, after line 11 is executed, $\Delta_{\mathbb{C}}(\theta') = \Delta_{\mathbb{C}}(\max(\theta']_{\Theta_{\mathbb{C}}})$; moreover, none of these will be changed anymore until line 16 is reached, which proves our result.

3. Since Γ is updated according to Δ in both $\mathbb{C}\langle X\rangle$ and $\mathbb{B}\langle X\rangle$, it is enough to prove that $\Delta^e_{\mathbb{C}} = \Delta_{\mathbb{B}}(\tau e)$. For $\mathbb{B}\langle X\rangle$, we have

1) $Dom(\Delta_{\mathbb{B}}(\tau e)) = \{\bot, \theta\} \sqcup \Theta_{\mathbb{C}} = (\{\theta\} \sqcup \Theta_{\mathbb{C}}) \cup \Theta_{\mathbb{C}}$;

2) $\forall\ \theta' \in \{\theta\} \sqcup \Theta_{\mathbb{C}}$, $\Delta_{\mathbb{B}}(\tau e)(\theta') = \sigma(\Delta_{\mathbb{B}}(\tau)(\max, (\theta']_{\Theta_{\mathbb{C}}}), e)$;

3) $\forall\ \theta' \in \Theta_{\mathbb{C}} - \{\theta\} \sqcup \Theta_{\mathbb{C}}$, $\Delta_{\mathbb{B}}(\tau e)(\theta') = \Delta_{\mathbb{B}}(\tau)(\theta')$.

So we only need to prove that

1) $Dom(\Delta^e_{\mathbb{C}}) = \{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}$;

2) $\forall\ \theta' \in \{\theta\} \sqcup \Theta_{\mathbb{C}}$, $\Delta^e_{\mathbb{C}}(\theta') = \sigma(\Delta^b_{\mathbb{C}}(\max, (\theta']_{\Theta_{\mathbb{C}}}), e)$;

3) $\forall\ \theta' \in \Theta_{\mathbb{C}} - \{\theta\} \sqcup \Theta_{\mathbb{C}}$, $\Delta^e_{\mathbb{C}}(\theta') = \Delta^b_{\mathbb{C}}(\theta')$.

By *1.*, we have $Dom(\Delta^m_{\mathbb{C}}) = \{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}$. Since lines 16 to 19 do not change $Dom(\Delta_{\mathbb{C}})$, $Dom(\Delta^e_{\mathbb{C}}) = Dom(\Delta^m_{\mathbb{C}}) = \{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}$. 1) holds.

By *2.* and Lemma 1, $\Delta^m_{\mathbb{C}}(\theta') = \Delta^b_{\mathbb{C}}(\max, (\theta']_{\Theta_{\mathbb{C}}})$ for any $\theta' \in Dom(\Delta^m_{\mathbb{C}})$. Also, notice that line 17 sets $\Delta_{\mathbb{C}}(\theta')$ to $\sigma(\Delta_{\mathbb{C}}(\theta'), e)$, which is $\sigma(\Delta^b_{\mathbb{C}}(\max, (\theta']_{\Theta_{\mathbb{C}}}), e)$, for the θ' in the loop. So, to show 2) and 3), we only need to prove that the loop at line 16 to 19 iterates over $\{\theta\} \sqcup \Theta_{\mathbb{C}}$. Since lines 16 to 19 do not change $\mathcal{U}_{\mathbb{C}}$, we need to show $\{\theta\} \cup \mathcal{U}^m_{\mathbb{C}}(\theta) = \{\theta\} \sqcup \Theta_{\mathbb{C}}$. Since $Dom(\Delta^m_{\mathbb{C}}) = \{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}$, we have $\{\theta\} \sqcup Dom(\Delta^m_{\mathbb{C}}) = \{\theta\} \sqcup (\{\bot, \theta\} \sqcup \Theta_{\mathbb{C}}) = \{\theta\} \sqcup ((\{\theta\} \sqcup \Theta_{\mathbb{C}}) \cup \Theta_{\mathbb{C}})$. By Proposition 4, $\{\theta\} \sqcup Dom(\Delta^m_{\mathbb{C}}) = (\{\theta\} \sqcup (\{\theta\} \sqcup \Theta_{\mathbb{C}})) \cup (\{\theta\} \sqcup \Theta_{\mathbb{C}}) = (\{\theta\} \sqcup \Theta_{\mathbb{C}}) \cup (\{\theta\} \sqcup \Theta_{\mathbb{C}}) = \{\theta\} \sqcup \Theta_{\mathbb{C}}$. Also, as $\theta \in Dom(\Delta^m_{\mathbb{C}})$, we have $\{\theta\} \sqcup Dom(\Delta^m_{\mathbb{C}}) = \{\theta' \mid \theta' \in Dom(\Delta^m_{\mathbb{C}}) \text{ and } \theta \sqsubseteq \theta'\} = \{\theta\} \sqcup \mathcal{U}^m_{\mathbb{C}}(\theta)$ by

Lemma 1. So $\{\theta\} \cup \mathcal{U}^m_{\mathbb{C}}(\theta) = \{\theta\} \sqcup \Theta_{\mathbb{C}}$. □

4.4 Optimized Online Parametric Monitoring

In this section, we show how to optimize parametric monitoring using extra information that can be easily extracted from specified properties.

4.4.1 Monitoring with Creation Events : $\mathbb{C}^+\langle X \rangle$

The monitor creation events are those events that are the first event in matching traces. The point of monitor creation events is to delay the expensive creation of monitor instances until a point where a pattern can actually be matched. For instance, using the UnsafeMapIterator example, there is no way a trace beginning with update_map$\langle m_1 \rangle$ can ever match the patterns or validate the formulae, so creating a monitor instance for m_1 is a waste of both time and memory.

The first challenge to online monitoring of a parametric property is that the state space of potential parameter instances is infinite. We encode partial functions $[[X \rightharpoonup V] \rightharpoonup Y]$, which map some parameter instances $[X \rightharpoonup V]$ to elements in Y, as tables with entries indexed by parameter instances in $[X \rightharpoonup V]$ and with elements in Y. It can be easily seen that, in what follows, such tables will have a finite number of entries provided that each event instantiates a finite number of parameters, which is always the case.

Figure 4.7 shows the algorithm $\mathbb{C}^+\langle X \rangle$ for online monitoring of parametric property $\Lambda X.P$, given that M is a monitor for P. The algorithm shows which actions to perform, e.g., creating a new monitor state and/or updating the state of related monitors, when an event is received. It is a slightly different variant of algorithm $\mathbb{C}\langle X \rangle$ in [31]. $\mathbb{C}^+\langle X \rangle$ is justified and motivated by experience with implementing and evaluating $\mathbb{C}\langle X \rangle$ in Section 4.3, mainly by the following observation: one often chooses to starting monitoring at the witness of a specific set of events (instead of monitoring from the beginning of the program). For example, when we monitor the UnsafeMapIterator property, we can choose to start monitoring on a pair of m and c objects, (m_1, c_1), only when a create_coll event is received, ignoring all the update_map$\langle m_1 \rangle$ events before the creation. We call such events that lead to creation of new monitor states *(monitor) creation events*. Algorithm $\mathbb{C}^+\langle X \rangle$ extends $\mathbb{C}\langle X \rangle$ in [31] to support creation events. It is easy to see that $\mathbb{C}\langle X \rangle$ can be regarded as a special case of $\mathbb{C}^+\langle X \rangle$, when all the events are creation events. Note that [31] used creation events in the evaluation, but they were not formalized in the algorithm. The proof of $\mathbb{C}^+\langle X \rangle$ is tedious, but is

easily derived from the above proof of $\mathbb{C}\langle X\rangle$.

```
Algorithm ℂ⁺⟨X⟩(M = (S, ℰ, 𝒞, i, σ, γ))
Globals: mapping Δ : [[X→V]→S]
         mapping 𝒰 : [X→V] → 𝒫_f([X→V])
Initialization: 𝒰(θ) ← ∅ for any θ ∈ [X→V]

function main(e⟨θ⟩)
 1  if Δ(θ) undefined then
 2  ⋮ for all θ_m ⊏ θ (in reversed topological order) do
 3  ⋮ ⋮ if Δ(θ_m) defined then
 4  ⋮ ⋮ ⋮ goto 7
 5  ⋮ ⋮ endif
 6  ⋮ endfor
 7  ⋮ if Δ(θ_m) defined then
 8  ⋮ ⋮ defineTo(θ, θ_m)
 9  ⋮ elseif e is a creation event then
10  ⋮ ⋮ defineNew(θ)
11  ⋮ endif
12  ⋮ for all θ_m ⊏ θ (in reversed topological order) do
13  ⋮ ⋮ for all θ_comp ∈ 𝒰(θ_m) compatible with θ do
14  ⋮ ⋮ ⋮ if Δ(θ_comp ⊔ θ) undefined then
15  ⋮ ⋮ ⋮ ⋮ defineTo(θ_comp ⊔ θ, θ_comp)
16  ⋮ ⋮ ⋮ endif
17  ⋮ ⋮ endfor
18  ⋮ endfor
19  endif
20  for all θ' ∈ {θ} ∪ 𝒰(θ) do
21  ⋮ Δ(θ') ← σ(Δ(θ'), e)
22  endfor
function defineNew(θ)
1  Δ(θ) ← i
2  for all θ'' ⊏ θ do
3  ⋮ 𝒰(θ'') ← 𝒰(θ'') ∪ {θ}
4  endfor
function defineTo(θ, θ')
1  Δ(θ) ← Δ(θ')
2  for all θ'' ⊏ θ do
3  ⋮ 𝒰(θ'') ← 𝒰(θ'') ∪ {θ}
4  endfor
```

Figure 4.7: Monitoring Algorithm $\mathbb{C}^+\langle X\rangle$.

Two mappings are used: Δ and $\mathcal{U}$. Δ stores the monitor states for parameter instances , and $\mathcal{U}$ maps a parameter instance θ to *all the parameter instances* that have been defined and are properly more informative than θ. In what follows, "the monitor state for θ" refers to $\Delta(\theta)$ to facilitate reading in some contexts, and, accordingly, "to create a parameter instance θ" and "to create a monitor state for parameter instance θ" have the same meaning: to define $\Delta(\theta)$.

When parametric event $e\langle\theta\rangle$ arrives, the algorithm first checks whether θ has been encountered

yet by checking if its corresponding monitor state, i.e., $\Delta(\theta)$, has been defined(line 1 in main). If θ is encountered for the first time, new parameter instances may need be created. In such a case, we first try to locate the maximum parameter instance (θ_m) which is less informative than θ and for which a monitor state has been created (lines 2 - 6). If such θ_m is found, its monitor state is used to initialize the monitor state for θ (lines 7 and 8); otherwise, a new monitor state is created for θ *only if* e is a creation event (lines 9 and 10). Also, new parameter instances can be created by combining θ with existing parameter instances that are compatible with θ, i.e., they do not have conflicting parameter bindings. An observation here is that if parameter instance θ_{comp} has been created and is compatible with θ then θ_{comp} can be found in $\mathcal{U}(\theta_m)$ for some $\theta_m \sqsubset \theta$ according to the definition of $\mathcal{U}$. Therefore, algorithm $\mathbb{C}^+\langle X\rangle$ searches through all the $\theta_m \sqsubset \theta$ to find all possible θ_{comp}, examining whether any new parameter instance should be created (lines 12 - 17).

If θ has been seen before, or otherwise after all the new monitor states have been created/initialized as explained above, algorithm $\mathbb{C}^+\langle X\rangle$ invokes all the monitors that need to process e, namely, those whose corresponding parameter instances are more informative than or equal to θ (lines 20 - 22). The updates make use of the sets stored in $\mathcal{U}$ to know which instances are more informative (line 20). There are two auxiliary functions: defineNew and defineTo. The former initializes a new monitor state for the input parameter instance and the latter creates a monitor state for the first input parameter instance using the monitor state for the second instance. Both functions add θ to the sets in table $\mathcal{U}$ for the bindings less informative than θ.

Event	update_map$\langle m_1\rangle$	create_coll$\langle m_1, c_1\rangle$	create_coll$\langle m_2, c_2\rangle$	create_iter$\langle c_1, i_1\rangle$
Δ	$\emptyset$	$\langle m_1, c_1\rangle$: $\sigma(i,$ create_coll$)$	$\langle m_1, c_1\rangle$: $\sigma(i,$ create_coll$)$ $\langle m_2, c_2\rangle$: $\sigma(i,$ create_coll$)$	$\langle m_1, c_1\rangle$: $\sigma(i,$ create_coll$)$ $\langle m_2, c_2\rangle$: $\sigma(i,$ create_coll$)$ $\langle m_1, c_1, i_1\rangle$: $\sigma(\sigma(i,$ create_coll$),$ create_iter$)$
$\mathcal{U}$	$\emptyset$	$\bot : \langle m_1, c_1\rangle$ $\langle m_1\rangle$: $\langle m_1, c_1\rangle$ $\langle c_1\rangle$: $\langle m_1, c_1\rangle$	$\bot : \langle m_1, c_1\rangle, \langle m_2, c_2\rangle$ $\langle m_1\rangle$: $\langle m_1, c_1\rangle$ $\langle c_1\rangle$: $\langle m_1, c_1\rangle$ $\langle m_2\rangle$: $\langle m_2, c_2\rangle$ $\langle c_2\rangle$: $\langle m_2, c_2\rangle$	$\bot : \langle m_1, c_1\rangle, \langle m_2, c_2\rangle, \langle m_1, c_1, i_1\rangle$ $\langle m_1\rangle$: $\langle m_1, c_1\rangle, \langle m_1, c_1, i_1\rangle$ $\langle c_1\rangle$: $\langle m_1, c_1\rangle, \langle m_1, c_1, i_1\rangle$ $\langle m_2\rangle$: $\langle m_2, c_2\rangle$ $\langle c_2\rangle$: $\langle m_2, c_2\rangle$ $\langle i_1\rangle$: $\langle m_2, c_2\rangle, \langle m_2, c_2, i_1\rangle$ $\langle m_1, c_1\rangle$: $\langle m_1, c_1, i_1\rangle$ $\langle m_1, i_1\rangle$: $\langle m_1, c_1, i_1\rangle$ $\langle c_1, i_1\rangle$: $\langle m_1, c_1, i_1\rangle$

Figure 4.8: Sample Run of $\mathbb{C}^+\langle X\rangle$. The first row gives the received events; the second and the third rows give the content of Δ and $\mathcal{U}$, respectively, after every event is processed. Monitor states are represented symbolically in the table, e.g., $\sigma(i,$ create_coll$)$ represents the state after a monitor processes event create_coll.

We next use an example run, illustrated in Figure 4.8, to show how $\mathbb{C}^+\langle X\rangle$ works. In Figure 4.8, we show the contents of Δ and $\mathcal{U}$ after every event (given in the first row of the table) is processed.

The observed trace is update_map$\langle m_1 \rangle$ create_coll$\langle m_1, c_1 \rangle$ create_coll $\langle m_2, c_2 \rangle$ create_iter$\langle c_1, i_1 \rangle$. We assume that create_coll is the only creation event.

The first event, update_map$\langle m_1 \rangle$, is not a creation event and nothing is added to Δ and $\mathcal{U}$. The second event, create_coll$\langle m_1, c_1 \rangle$, is a creation event. So a new monitor state is defined in Δ for $\langle m_1, c_1 \rangle$, which is also added to the lists in $\mathcal{U}$ for $\perp$, $\langle m_1 \rangle$ and $\langle c_1 \rangle$. Note that $\perp$ is less informative than any other parameter instances. The third event create_coll$\langle m_2, c_2 \rangle$ is another creation event, incompatible with the second event . Hence, only one new monitor state is added to Δ. $\mathcal{U}$ is updated similarly. The last event create_iter$\langle c_1, i_1 \rangle$ is not a creation event. So no monitor instance is created for $\langle c_1, i_1 \rangle$. It is compatible with the existing parameter instance $\langle m_1, c_1 \rangle$ introduced by the second event but not compatible with $\langle m_2, c_2 \rangle$ due to the conflict binding on c. The compatible instance $\langle m_1, c_1 \rangle$ can be found from the list for $\langle c_1 \rangle$ in $\mathcal{U}$. Therefore, a new monitor instance is created for the combined parameter instance $\langle m_1, c_1, i_1 \rangle$ using the state for $\langle m_1, c_1 \rangle$ in Δ. $\mathcal{U}$ is also updated to add the combined parameter instance into lists of parameter instances that are less informative.

4.4.2 Limitations of $\mathbb{C}^+\langle X \rangle$ and Enable Sets

$\mathbb{C}^+\langle X \rangle$ does not make any assumption on the given monitor M. In other words, one may monitor properties written in any specification formalism, e.g., ERE, CFG, PTLTL etc., as long as one also provides a monitor generation algorithm for said formalism. However, this generality leads to extra monitoring overhead in some cases. Thus we introduce our novel optimization based on the concept of enable sets.

To motivate the optimization, let us continue the run in Figure 4.8 to process one more event, use_iter$\langle i_1 \rangle$. The result is shown in Figure 4.9. use_iter$\langle i_1 \rangle$ is not a creation event and no monitor instance is created for $\langle i_1 \rangle$. Since $\langle i_1 \rangle$ is compatible with $\langle m_2, c_2 \rangle$, a new monitor instance is defined for $\langle m_2, c_2, i_1 \rangle$. The monitor instance for $\langle m_1, c_1, i_1 \rangle$ is then updated according to use_iter because $\langle i_1 \rangle$ is less informative than $\langle m_1, c_1, i_1 \rangle$. $\mathcal{U}$ is also updated to add $\langle m_2, c_2, i_1 \rangle$ to the lists for all the parameter instances less informative than $\langle m_2, c_2, i_1 \rangle$. New entries are added into $\mathcal{U}$ during the update since some of less informative parameter instances, e.g., $\langle m_2, i_1 \rangle$, have not been used before this event.

Creating the monitor instance for $\langle m_2, c_2, i_1 \rangle$ is needed for the correctness of $\mathbb{C}^+\langle X \rangle$, but it can be avoided when more information about the program or the specification is available. For example, according to the semantics of Iterator, no event create_iter$\langle c_2, i_1 \rangle$ will occur in the following execution since an iterator can be associated to only one collection. Hence, the monitor for $\langle m_2, c_2, i_1 \rangle$ will

Event	use_iter$\langle i_1 \rangle$
Δ	$\langle m_1, c_1 \rangle$: $\sigma(i, \text{create\_coll})$ $\langle m_2, c_2 \rangle$: $\sigma(i, \text{create\_coll})$ $\langle m_1, c_1, i_1 \rangle$: $\sigma(\sigma(\sigma(i, \text{create\_coll}), \text{create\_iter}), \text{use\_iter})$ $\langle m_2, c_2, i_1 \rangle$: $\sigma(\sigma(i, \text{create\_coll}), \text{use\_iter})$
$\mathcal{U}$	$\bot : \langle m_1, c_1 \rangle, \langle m_2, c_2 \rangle, \langle m_2, c_2, i_1 \rangle, \langle m_1, c_1, i_1 \rangle$ $\langle m_1 \rangle$: $\langle m_1, c_1 \rangle, \langle m_1, c_1, i_1 \rangle$ $\langle c_1 \rangle$: $\langle m_1, c_1 \rangle, \langle m_1, c_1, i_1 \rangle$ $\langle m_2 \rangle$: $\langle m_2, c_2 \rangle, \langle m_2, c_2, i_1 \rangle$ $\langle c_2 \rangle$: $\langle m_2, c_2 \rangle, \langle m_2, c_2, i_1 \rangle$ $\langle i_1 \rangle$: $\langle m_2, c_2, i_1 \rangle, \langle m_1, c_1, i_1 \rangle$ $\langle m_2, \text{c} \mapsto c_2 \rangle$: $\langle m_2, c_2, i_1 \rangle$ $\langle m_2, i_1 \rangle$: $\langle m_2, c_2, i_1 \rangle$ $\langle c_2, i_1 \rangle$: $\langle m_2, c_2, i_1 \rangle$ $\langle m_1, c_1 \rangle$: $\langle m_1, c_1, i_1 \rangle$ $\langle m_1, i_1 \rangle$: $\langle m_1, c_1, i_1 \rangle$ $\langle c_1, i_1 \rangle$: $\langle m_1, c_1, i_1 \rangle$

Figure 4.9: Following the Run of Figure 4.8.

never reach the validation state and we do not need to create it from the beginning. However, such semantic information about the program is very difficult to infer automatically. Below, we show a simpler yet effective solution to avoid unnecessary monitor creations by analyzing the specification to monitor.

When monitoring a program against a specific property, usually only a certain subset of property categories, ($\mathcal{C}$ in Definition 5), is checked. For example, the regular expression for the UnsafeMapIterator specifies a defective interaction among related Map, Collection and Iterator objects. To find an error in the program using monitoring is thus to detect matches of the specified pattern during the execution. In other words, we are only interested in the validation category of the specified pattern. Obviously, to match the pattern, for a parameter instance of parameter set {m, c, i}, create_coll and create_iter should be observed before use_iter is encountered for the first time in monitoring. Otherwise, the trace slice for {m, c, i} will never match the pattern. Based on this information, we next show that creating the monitor state for $\langle m_2, c_2, i_1 \rangle$ in Figure 4.9 is not needed. When event use_iter$\langle i_1 \rangle$ is encountered, if the monitor state for a parameter instance $\langle m_2, c_2 \rangle$ exists without the monitor state for $\langle m_2, c_2, i_1 \rangle$, like in Figure 4.9, it can be inferred that in the trace slice for $\langle m_2, c_2, i_1 \rangle$, only events create_coll and/or update_map occur before use_iter because, otherwise, if create_iter also occurred before use_iter, the monitor state for $\langle m_2, c_2, i_1 \rangle$ should have been created. Therefore, we can infer, when event use_iter$\langle i_1 \rangle$ is observed and before the execution continues, that no match of the specified pattern can be reached by the trace slice for $\langle m_2, c_2, i_1 \rangle$, that is to say, the monitor for $\langle m_2, c_2, i_1 \rangle$ will never reach the validation state.

This observation shows that the knowledge about the specified property can be applied to avoid

Event	$\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(\text{Event})$
create_coll	$\{\emptyset\}$
create_iter	$\{\{\text{create\_coll}\},$ $\{\text{create\_coll, update\_map}\}\}$
use_iter	$\{\{\text{create\_coll, create\_iter}\},$ $\{\text{create\_coll, create\_iter, update\_map}\}\}$
update_map	$\{\{\text{create\_coll}\},$ $\{\text{create\_coll, create\_iter}\},$ $\{\text{create\_coll, create\_iter, use\_iter}\}\}$

Figure 4.10: Property Enable Set for UnsafeMapIterator.

unnecessary creation of monitor states. This way, the sizes of Δ and $\mathcal{U}$ can be reduced, reducing the monitoring overhead. We next formalize the information needed for the optimization and argue that it is not specific to the underlying specification formalism, and that it can be computed easily. How this information is used is discussed in Section 4.4.3.

Enable Sets

Definition 24 *Given $\tau \in \mathcal{E}^*$ and $e, e' \in \tau$, we denote that e' occurs before the first occurrence of e in τ as $e' \rightsquigarrow_\tau e$. Let the* ***trace enable set*** *of $e \in \mathcal{E}$ be the function $\mathsf{enable}_\tau : \mathcal{E} \to \mathcal{P}_f(\mathcal{E})$, defined as: $\mathsf{enable}_\tau(e) = \{e' \mid e' \rightsquigarrow_\tau e\}$.*

Note that if $e \notin \tau$ then $\mathsf{enable}_\tau(e) = \emptyset$. The trace enable set can be used to examine whether the execution under observation may generate a particular trace of interest, or not: if event e is encountered during monitoring but some event $e' \in \mathsf{enable}_\tau(e)$ has not been observed, then the (incomplete) execution being monitored will *not* produce the trace τ when it finishes. This observation can be extended to check, before an execution finishes, whether the execution can generate a trace belonging to some designated property categories. The designated property categories are called the *goal* of the monitoring in what follows.

Definition 25 *Given $P : \mathcal{E}^* \to \mathcal{C}$ and a set of categories $\mathcal{G} \subseteq \mathcal{C}$ as the goal, the* ***property enable set*** *is defined as a function $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(\mathcal{E}))$ with $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(e) = \{\mathsf{enable}_\tau(e) \mid P(\tau) \in \mathcal{G}\}$.*

Intuitively, if event e is encountered during monitoring but none of event sets $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(e)$ has been completely observed, the (incomplete) execution being monitoring will not produce a trace τ s.t. $P(\tau) \in \mathcal{G}$. For example, given the UnsafeMapIterator property specified by ERE, where $\mathcal{G}$ contains only the match, violation, and ? categories, Figure 4.10 shows the property enable set for UnsafeMapIterator.

The property enable set provides a sound and fast way to decide whether an incomplete trace slice has the possibility of reaching the desired categories by looking at the events that have already occurred. In the above example, if a trace slice starts with create_coll use_iter, it will never reach the match category, because $\{\mathsf{create\_coll}\} \notin \mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(\mathsf{use\_iter})$. In such case, no monitor state need be created even when the newly observed event may lead to new parameter instances. For example, suppose that the observed (incomplete) trace is $\mathsf{create\_coll}\langle m_1, c_1\rangle$ $\mathsf{use\_iter}\langle i_1\rangle$. At the second event, $\mathsf{use\_iter}\langle i_1\rangle$, a new parameter instance can be constructed, namely, $\langle m_1, c_1, i_1\rangle$, and a monitor state s will be created for $\langle m_1, c_1, i_1\rangle$ if algorithm $\mathbb{C}^+\langle X\rangle$ is applied. However, since the trace slice for s is create_coll use_iter, we can immediately know that s cannot reach the match state, and thus there is no need to create and maintain s during monitoring if match is the target category.

A direct application of the above idea to optimize $\mathbb{C}^+\langle X\rangle$ requires maintaining observed events for every created monitor and comparing event sets when a new parameter instance is found, reducing the improvement of performance. Therefore, we extend the notion of the enable set to be based on parameter sets instead of event sets.

Definition 26 *Given a property $P : \mathcal{E}^* \to \mathcal{C}$, a set of categories $\mathcal{G} \subseteq \mathcal{C}$ as the goal, a set of parameters X and a parameter definition $\mathcal{D}_{\mathcal{E}}$, the* ***property parameter enable set*** *of event $e \in \mathcal{E}$ is defined as a function $\mathsf{enable}^X_{\mathcal{G}} : \mathcal{E} \to \mathcal{P}_f(\mathcal{P}_f(X))$ as follows: $\mathsf{enable}^X_{\mathcal{G}}(e) = \{\cup\{\mathcal{D}_{\mathcal{E}}(e') \mid e' \in \mathsf{enable}_\tau(e)\} \mid P(\tau) \in \mathcal{G}\}$.*

From now on, we use "enable set" to refer to "property parameter enable set" for simplicity. For example, given the ERE-based UnsafeMapIterator property and $\mathcal{G} = \{\mathsf{validating}\}$; Figure 4.11 shows the parameter enable set for UnsafeMapIterator. Then, given again the trace $\{\mathsf{create\_coll}\}\langle m_1, c_1\rangle$ $\mathsf{use\_iter}\langle i_1\rangle$, no monitor state need be created at the second event for $\langle m_1, c_1, i_1\rangle$ since the parameter instance used to initialize the new monitor state, namely, $\langle m_1, c_1\rangle$, is not in $\mathsf{enable}^X_{\mathcal{G}}(\mathsf{use\_iter})$. In other words, one may simply compare the parameter instance used to initialize the new parameter instance with the enable set of the observed event to decide whether a new monitor state is needed or not. Note that in JavaMOP, the property parameter enable sets are generated from the property enable sets provided by the formalism plugin in question. This allows the plugins to remain totally parameter agnostic. The following result guarantees the correctness of this approach:

Proposition 17 *When algorithm $\mathbb{C}^+\langle X\rangle$ receives event $e\langle\theta\rangle$, if we use θ' to define $\theta \sqcup \theta'$ and $Dom(\theta') \notin \mathsf{enable}^X_{\mathcal{G}}(e)$, then $\Delta(\theta \sqcup \theta') \notin \mathcal{G}$ during the whole monitoring process.*

Event	$\mathsf{enable}_{\mathcal{G}}^{X}(\text{Event})$
create_coll	$\{\emptyset\}$
create_iter	$\{\{\mathsf{m}, \mathsf{c}\}\}$
use_iter	$\{\{\mathsf{m}, \mathsf{c}, \mathsf{i}\}\}$
update_map	$\{\{\mathsf{m}, \mathsf{c}\}, \{\mathsf{m}, \mathsf{c}, \mathsf{i}\}\}$

Figure 4.11: Parameter Enable Set for UnsafeMapIterator.

Computing Enable Sets

The definition of the enable set is general and does not depend on a specific formalism to write the property. Although computing the enable set from a specified property requires understanding of the used formalism. It can be achieved as a "side-effect" of the monitor generation process, in which full knowledge about the property is available.

```
Algorithm EN_fsm(FSM = (E, S, s_0, δ, F))
Globals: mapping V_μ : S → P_f(P_f(E))
         mapping enable_G^E : E → P_f(P_f(E))
         set R ⊆ S

Initialization: fix G' ⊆ S, compute R for G'

function main()
 1  auxiliary(s_0, ∅)
function auxiliary(s, μ)
 1  for all e ∈ E do
 2  ⋮ if δ(s, e) ∈ R then
 3  ⋮ ⋮ enable_G^E(e) ← enable_G^E(e) ∪ {μ − e}
 4  ⋮ endif
 5  ⋮ let μ' ← μ ∪ {e}
 6  ⋮ ⋮ if μ' ∉ V_μ(s)
 7  ⋮ ⋮ ⋮ V_μ(s) ← V_μ(s) ∪ {μ'}
 8  ⋮ ⋮ ⋮ auxiliary(δ(s, e), μ')
 9  ⋮ ⋮ endif
10 endfor
```

Figure 4.12: FSM Enable Set Computation Algorithm.

Case 1: FSM The algorithm in Figure 4.12 computes the property enable sets for a finite state machine. We use this algorithm to compute the enable sets for any logic that is reducible to a finite state machine, including ERE, PTLTL, and FTLTL. The algorithm assumes a finite state machine, defined as $FSM = (\mathcal{E}, S, s_0 \in S, \delta : S \times \mathcal{E} \rightarrow S, F \subseteq S)$. $\mathcal{E}$ is the alphabet, traditionally listed as Σ but changed for consistency, since the alphabets of our FSMs are event sets. s_0 is the start state, corresponding to i in the definition of a monitor. δ is the transition function, taking a state

and an event and mapping to a next state for the machine. F is the set of accept states. In the initialization we compute goal reachability set S by fixing a goal G' as an arbitrary set of states, such as the error state for violation, or accept states for matching a pattern originally specified as an ERE. More specifically, G' is the subset of S corresponding to the subset of $\mathcal{G}$ in which we are interested. For state $s \in S$, $s \in R$ if and only if there is a path from s to an $s' \in G'$. It is computed using a straight-forward depth first search from the initial state. $\mathcal{V}_\mu$ is a mapping from states to sets of events; it is used to check for algorithm termination. $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$ is the output property enable set, which is converted into a parameter enable set by JavaMOP.

Function auxiliary is first called with $\mu = \emptyset$ and the initial state s_0 (the Initialization section). If we think of the FSM as a graph, μ represents the set of edges we have seen at least once in a traversal. For each event in $\mathcal{E}$ (line 1), we check to see if the next state, computed by $\delta(s, e)$ reaches our goal (line 2). If it does, that means we have seen a viable prefix set. From the definition of $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$, we know we need to add this prefix set to $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$ for the event e, which we do (line 3). Also on line 3, we make sure that we remove e from μ, as an enable set for e is not supposed to contain e. Line 5 begins the recursive step of the algorithm. We let $\mu' = \mu \cup \{e\}$, because we have traversed another edge, and that edge is labeled as e. The map $\mathcal{V}_\mu$ tells us which μ have been seen in previous recursive steps, in a given state. If a μ has been seen before, in a state, taking a recursive step can add no new information. Because of this, line 6 ensures that we only call the recursive step on line 8, if new information can be added. Line 7 keeps $\mathcal{V}$ consistent. Thus the algorithm terminates only when every viable μ has been seen in every reachable state, effectively computing a fixed point.

Case 2: CFG We also provide an algorithm to compute the enable set for a context-free pattern, which has an infinite monitor state space, as briefly explained in what follows[2]. This is a modification of the algorithm in Figure 4.12.

Let $\mathcal{G} = \{\mathsf{match}\}$. For $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}$ and a given CFG $G = (NT, \mathcal{E}, P, S)$ we begin with all productions $S \to \gamma$ and the set $\mu_0 = \emptyset \in \mathcal{P}_f(\mathcal{E})$. For each production, we investigate each $s \in \gamma$ (where $\in$ is, by abuse of notation, used to denote a symbol in a right hand side) from left to right. If $s \in \mathcal{E}$ we add μ_i to $\mathsf{enable}^{\mathcal{E}}_{\mathcal{G}}(s)$, thus if s is the first symbol in γ we add μ_0. We then add s to μ_i forming μ_{i+1}. If $s \in NT$ we recursively invoke the algorithm, but rather than use μ_0, we use μ_i, and each production investigated will be of the form $s \to \gamma$. We keep track of which $s \in NT$ have been processed, to ensure termination.

Discussion. The general definition of the enable set allows us to separate the concerns of generating

[2]We assume a certain familiarity with context free patterns; definitions can be found in [67], together with explanations on CFG monitoring.

efficient monitoring code. On the framework level, such as the algorithms discussed in this paper, we can focus on applying the information encoded in the enable set to generate an efficient monitoring process for parametric properties, while on the logic level, where a monitor is generated for a given non-parametric property written in a specific formalism, one can focus on creating the fastest monitor that verifies the input trace against the property and also on producing the enable set information. The enable set represents static information about the given property and only need be generated once. As mentioned, the static analysis presented in [21], while effective, requires a complex analysis of the target program, which must be performed for every program one wants to monitor.

Other possibilities for optimization are exhibited in the example in Figure 4.9. We discuss two of them here. The first is to make use of the semantics of the program. In this example, we know that an i object is created from a c object and does not relate to other c objects. Hence, we can avoid creating a combination of $\langle m_2, c_2 \rangle$ and $\langle i_1 \rangle$ because i_1 is created from c_1. However, such semantic information is very difficult to achieve automatically and may require human input. The enable set, on the contrary, can be easily computed by statically analyzing the specification without analyzing any program or human interferences; indeed, the specified property already indicates some semantics of the involved parameters. Nevertheless, we believe that static analysis on the program to monitor, such as that in [21], can and should be applied in conjunction with enable sets to further reduce the monitoring overhead, whenever it is feasible.

Other optimizations are based on heuristics. One reasonable heuristic which can be applied here is that we may only combine parameter instances that are connected to one another through some events which have been observed (we cannot rely on future events in online monitoring). For example, $\langle i_1 \rangle$ and $\langle m_1, c_1 \rangle$ need to be combined to build a new parameter instance because c_1 and i_1 are connected in the second event, $\mathsf{create\_coll}\langle m_1, c_1 \rangle$, in Figure 4.9, but $\langle i_1 \rangle$ and $\langle m_2, c_2 \rangle$ should not combined due to the heuristic. The intuition is that if two parameter instances do not interact in any event, it may imply that they are not relevant to each other even if they are compatible. However, because no information about future events available, such a heuristic can break, for example, an event connecting the two parameter instances comes afterward. The enable set provides a sound optimization, and we believe that it performs as well as, if not better than, such heuristics in most cases.

4.4.3 Monitoring with Enable Sets: $\mathbb{D}\langle X\rangle$

In this section we integrate the concept of enable sets with algorithm $\mathbb{C}^+\langle X\rangle$, to improve performance and memory usage. To ease reading, all proofs related to this algorithm can be found in Section 4.4.3.

Given a set of desired value categories $\mathcal{G}$, Proposition 17 guarantees that we can omit creating monitor states for certain parameter instances when an event is received using the enable set without missing any trace belonging to $\mathcal{G}$. However, skipping the creation of monitor states may result in false alarms, i.e., a trace that is not in $\mathcal{G}$ can be reported to belong to $\mathcal{G}$. Let us consider the following example. We monitor to find matching of a regular pattern e_1e_3 and the event definition is $(e_1{\mapsto}\{P_1\}, e_2{\mapsto}\{P_2\}, e_3{\mapsto}\{P_1, P_2\})$ the observed trace is $e_1\langle p_1\rangle e_2\langle p_2\rangle e_3\langle p_1, p_2\rangle$. Also, suppose e_1 is the only creation event. Obviously, the trace does not match the pattern. Figure 4.13 shows the run using the optimization based on the enable set. Only the content of Δ is given for simplicity. At e_1, a monitor state is created for $\langle p_1\rangle$ since it is the creation event. At e_2, no action is taken since $\mathsf{enable}^X_{\mathcal{G}}(e_2) = \emptyset$. At e_3, a monitor state will be created for $\langle p_1, p_2\rangle$ using the monitor state for $\langle P_1 \mapsto p_1\rangle$ since $\mathsf{enable}^X_{\mathcal{G}} e_3 = \{P_1\}$. This way, e_2 is forgotten and a match of the pattern is reported even though it is not correct to do so.

Event	$e_1\langle p_1\rangle$	$e_2\langle p_2\rangle$	$e_3\langle p_1, p_2\rangle$
Δ	$\langle p_1\rangle{:}\sigma(i, e_1)$	$\langle p_1\rangle{:}\sigma(i, e_1)$	$\langle p_1\rangle{:}\sigma(i, e_1)$ $\langle p_1, p_2\rangle{:}\sigma(\sigma(i, e_1), e_3)$

Figure 4.13: Unsound Usage of the Enable Set.

Timestamping Monitors: Algorithm $\mathbb{D}\langle X\rangle$

To avoid unsoundness, we introduce the notion of $\mathsf{disable}$ stamps of events. $\mathsf{disable} : [[X \rightharpoonup V] \rightharpoonup \mathsf{integer}]$ maps a parameter instance to an integer timestamp. $\mathsf{disable}(\theta)$ gives the time when the last event with θ was received. We maintain timestamps for monitors using a mapping $\mathcal{T} : [[X \rightharpoonup V] \rightharpoonup \mathsf{integer}]$. $\mathcal{T}$ maps a parameter instance for which a monitor state is defined to the time when the original monitor state is created from a creation event. Specifically, if a monitor state for θ is created using the initial state when a creation event is received (i.e., using the $\mathsf{defineNew}$ function in algorithm $\mathbb{C}^+\langle X\rangle$), $\mathcal{T}(\theta)$ is set to the time of creation; if a monitor state for θ is created from the monitor state for θ', $\mathcal{T}(\theta')$ is passed to $\mathcal{T}(\theta)$. Figure 4.14 shows the evolution of $\mathsf{disable}$ and $\mathcal{T}$ while processing the trace in Figure 4.13.

$\mathsf{disable}$ and $\mathcal{T}$ can be used together to track "skipped events": when a monitor state for θ is created

using the monitor state for θ', if there exists some $\theta'' \sqsubset \theta$ s.t. $\theta'' \not\sqsubseteq \theta'$ and $\mathsf{disable}(\theta'') > \mathcal{T}(\theta')$ then the trace slice for θ does not belong to the desired value categories $\mathcal{G}$. Intuitively, $\mathsf{disable}(\theta'') > \mathcal{T}(\theta')$ implies that an event $e\langle\theta''\rangle$ has been encountered after the monitor state for θ' was created. But θ'' was not taken into account ($\theta'' \not\sqsubseteq \theta'$). The only possibility is that e is omitted due to the enable set and thus the trace slice for θ does not belong to $\mathcal{G}$ according to the definition of the enable set. Therefore, in Figure 4.14, no monitor instance is created for $\langle p_1, p_2\rangle$ at e_3 because $\mathsf{disable}(\langle p_2\rangle) > \mathcal{T}(\langle p_1\rangle)$.

Event	$e_1\langle p_1\rangle$	$e_2\langle p_2\rangle$	$e_3\langle p_1, p_2\rangle$
Δ	$\langle p_1\rangle$:$\sigma(i, e_1)$	$\langle p_1\rangle$:$\sigma(i, e_1)$	$\langle p_1\rangle$:$\sigma(i, e_1)$
$\mathcal{T}$	$\langle p_1\rangle$:1	$\langle p_1\rangle$:1	$\langle p_1\rangle$:1
disable	$\langle p_1\rangle$:2	$\langle p_1\rangle$:2 $\langle p_2\rangle$:3	$\langle p_1\rangle$:2 $\langle p_2\rangle$:3 $\langle p_1, p_2\rangle$:4

Figure 4.14: Sound Monitoring Using Enable Sets and Timestamps.

The above discussion applies when the skipped event occurs after the initial creation of the monitor state. The other case, i.e., an event is omitted before the initial monitor state is created, can also be handled using timestamps. First, if the skipped event is not a creation event, it does not affect the soundness of the algorithm to omit the event because of the definition of creation events. In the above example, if the observed trace is $e_2\langle p_2\rangle e_1\langle p_1\rangle e_3\langle p_1, p_2\rangle$, we will ignore e_2 and report the matching at e_3 since e_1 is the only creation event. The situation becomes more sophisticated when the skipped event is a creation event. For example, we assume that both e_1 and e_2 are creation events in the above example. Figure 4.15 then shows the monitoring process for the parametric trace $e_2\langle p_2\rangle e_1\langle p_1\rangle e_3\langle p_1, p_2\rangle$.

At e_2, $\Delta(\langle p_2\rangle)$ is defined because it is a creation event. At e_1, $\Delta(\langle p_1\rangle)$ is defined, but no monitor state is created for $\langle p_1, p_2\rangle$ because $\{P_2\} \notin \mathsf{enable}^X_{\mathcal{G}}(e_1)$. At e_3, we cannot use $\Delta(\langle p_2\rangle)$ to define $\Delta(\langle p_1, p_2\rangle)$ since $\mathsf{disable}(\langle p_1\rangle) > \mathcal{T}(\langle p_2\rangle)$. Moreover, we cannot use $\Delta(\langle p_1\rangle)$ to define $\Delta(\langle p_1, p_2\rangle)$, either, because $\Delta(\langle p_2\rangle)$ was defined before $\Delta(\langle p_1\rangle)$ but was not used to create $\Delta(\langle p_1, p_2\rangle)$ at e_1 due to the use of the enable set, indicating that the trace slice for $\langle p_1, p_2\rangle$ does not belong to $\mathcal{G}$, and it should be ignored during monitoring. This intuition can be captured as the following condition: $\mathcal{T}(\langle p_2\rangle) < \mathcal{T}(\langle p_1\rangle)$ and $\langle p_2\rangle \not\sqsubseteq \langle p_1\rangle$. To reiterate, if $\Delta(\theta')$ is used to define $\Delta(\theta)$ and there exists some $\theta'' \sqsubset \theta$ s.t. $\theta'' \not\sqsubseteq \theta'$ and $\mathcal{T}(\theta'') < \mathcal{T}(\theta')$, then the trace slice for θ does not belong to the desired category set $\mathcal{G}$, because θ would have been in the enable set of θ' if it were in $\mathcal{G}$. Such a situation happens at the following conditions: 1) a creation event, $e\langle\theta''\rangle$, is encountered before $\Delta(\theta')$ is defined

at event e'; 2) e is omitted when $\Delta(\theta')$ is defined (otherwise $\Delta(\theta'' \sqcup \theta')$ should have been defined and should be used to define θ instead of θ'). The second condition implies that $Dom(\theta'') \notin \mathsf{enable}_{\mathcal{G}}^{X}(e')$. Therefore, when we combine θ'' and θ' in θ, the trace slice for θ cannot belong to $\mathcal{G}$, due to the definition of enable set.

Event	$e_2\langle p_2\rangle$	$e_1\langle p_1\rangle$	$e_3\langle p_1, p_2\rangle$
Δ	$\langle p_2\rangle$:$\sigma(i, e_2)$	$\langle p_2\rangle$:$\sigma(i, e_2)$ $\langle p_1\rangle$:$\sigma(i, e_1)$	$\langle p_2\rangle$:$\sigma(i, e_2)$ $\langle p_1\rangle$:$\sigma(i, e_1)$
$\mathcal{T}$	$\langle p_2\rangle$:1	$\langle p_2\rangle$:1 $\langle p_1\rangle$:3	$\langle p_2\rangle$:1 $\langle p_1\rangle$:3
disable	$\langle p_2\rangle$:2	$\langle p_2\rangle$:2 $\langle p_1\rangle$:4	$\langle p_2\rangle$:2 $\langle p_1\rangle$:4 $\langle p_1, p_2\rangle$:5

Figure 4.15: Another Monitoring Using Enable Sets and Timestamps.

Based on the above discussion, we develop a new parametric monitoring algorithm that optimizes algorithm $\mathbb{C}^{+}\langle X\rangle$ using the enable set and timestamps, as shown in Figure 4.16. This algorithm makes use of the mappings discussed above, namely, $\mathsf{enable}_{\mathcal{G}}^{X}$, Δ, $\mathcal{U}$, disable and $\mathcal{T}$, and maintains an integer variable to track the timestamp. Similar to algorithm $\mathbb{C}^{+}\langle X\rangle$, when event $e\langle\theta\rangle$ is received, algorithm $\mathbb{D}\langle X\rangle$ first checks whether $\Delta(\theta)$ is defined or not (line 1 in main). If not, monitor states may be generated for new encountered parameter instances, which is achieved by function createNewMonitorStates in algorithm $\mathbb{D}\langle X\rangle$. Unlike in algorithm $\mathbb{C}^{+}\langle X\rangle$, where all the parameter instances less informative than θ are searched to find all the compatible parameter instances using $\mathcal{U}$, createNewMonitorStates enumerates parameter sets in $\mathsf{enable}_{\mathcal{G}}^{X}(e)$ and looks for parameter instances whose domains are in $\mathsf{enable}_{\mathcal{G}}^{X}(e)$ and which are compatible with θ, also using $\mathcal{U}$. The inclusion check at line 2 in createNewMonitorStates is to omit unnecessary search since if $Dom(\theta) \subseteq X_e$ then no new parameter instance will be created from θ. This way, createNewMonitorStates creates all the parameter instances that combine θ with compatible parameter instances that also satisfy the enable set of e using fewer lists in $\mathcal{U}$.

If e is a creation event then a monitor state for θ is initialized (lines 3 - 5 in main). Note that $\Delta(\theta)$ can be defined in function createNewMonitorStates if $\Delta(\theta')$ has been defined for some $\theta' \sqsubset \theta$. disable(θ) is set to the current timestamp after all the creations and the timestamp is increased. The rest of function main in $\mathbb{D}\langle X\rangle$ is the same as in $\mathbb{C}^{+}\langle X\rangle$: all the relevant monitor states are updated according to e.

Function defineNew in $\mathbb{D}\langle X\rangle$ is similar to the one in $\mathbb{C}^{+}\langle X\rangle$. The only difference is that $\mathcal{T}(\theta)$ is set to the current timestamp, and the timestamp is incremented. Function defineTo in $\mathbb{D}\langle X\rangle$ checks disable and $\mathcal{T}$ as discussed above to decide whether $\Delta(\theta)$ can be defined using $\Delta(\theta')$. If $\Delta(\theta)$ is

```
Algorithm 𝔻⟨X⟩(M = (S, ℰ, 𝒞, i, σ, γ))
Input: mapping enable_𝒢^X : [ℰ→𝒫_f(𝒫_f(X))]
Globals: mapping Δ : [[X→V]→S]
         mapping 𝒯 : [[X→V]→integer]
         mapping 𝒰 : [X→V] → 𝒫_f([X→V])
         mapping disable : [[X→V]→integer]
         integer timestamp
Initialization: 𝒰(θ) ← ∅ for any θ, timestamp ← 0

function main(e⟨θ⟩)
 1 if Δ(θ) undefined then
 2 ⋮ createNewMonitorState(e⟨θ⟩)
 3 ⋮ if Δ(θ) undefined and e is a creation event then
 4 ⋮ ⋮ defineNew(θ)
 5 ⋮ endif
 6 ⋮ disable(θ) ← timestamp
 7 ⋮ timestamp ← timestamp + 1
 8 endif
 9 for all θ' ∈ {θ} ∪ 𝒰(θ) s.t. Δ(θ') defined do
10 ⋮ Δ(θ') ← σ(Δ(θ'), e)
11 endfor
function createNewMonitorStates(e⟨θ⟩)
 1 for all X_e ∈ enable_𝒢^X(e)
     (in reversed topological order) do
 2 ⋮ if Dom(θ) ⊈ X_e then
 3 ⋮ ⋮ θ_m ← θ' s.t. θ' ⊏ θ and Dom(θ') = Dom(θ) ∩ X_e
 4 ⋮ ⋮ for all θ'' ∈ 𝒰(θ_m) ∪ {θ_m} s.t. Dom(θ'') = X_e do
 5 ⋮ ⋮ ⋮ if Δ(θ'') defined and Δ(θ'' ⊔ θ) undefined then
 6 ⋮ ⋮ ⋮ ⋮ defineTo(θ'' ⊔ θ, θ'')
 7 ⋮ ⋮ ⋮ endif
 8 ⋮ ⋮ endfor
 9 ⋮ endif
10 endfor
function defineNew(θ)
 1 for all θ'' ⊏ θ do
 2 ⋮ if Δ(θ'') defined then return endif
 3 endfor
 4 Δ(θ) ← i, 𝒯(θ) ← timestamp
 5 timestamp ← timestamp + 1
 6 for all θ'' ⊏ θ do
 7 ⋮ 𝒰(θ'') ← 𝒰(θ'') ∪ {θ}
 8 endfor
function defineTo(θ, θ')
 1 for all θ'' ⊑ θ s.t. θ'' ⋤ θ' do
 2 ⋮ if disable(θ'') > 𝒯(θ') or 𝒯(θ'') < 𝒯(θ') then
 3 ⋮ ⋮ return
 4 ⋮ endif
 5 endfor
 6 Δ(θ) ← Δ(θ'), 𝒯(θ) ← 𝒯(θ')
 7 for all θ'' ⊏ θ do 𝒰(θ'') ← 𝒰(θ'') ∪ {θ} endfor
```

Figure 4.16: Optimized Monitoring Algorithm $\mathbb{D}\langle X\rangle$.

defined using $\Delta(\theta')$, $\mathcal{T}(\theta)$ is set to $\mathcal{T}(\theta')$.

In all of our tested cases $\mathbb{D}\langle X\rangle$ performs better than $\mathbb{C}^+\langle X\rangle$; in fact, in most cases that caused notable monitoring overhead, the efficiency of $\mathbb{D}\langle X\rangle$ is significantly better than $\mathbb{C}^+\langle X\rangle$. For example, in two extreme cases, $\mathbb{C}^+\langle X\rangle$ could not finish, while $\mathbb{D}\langle X\rangle$ had no problems. In terms of memory usage $\mathbb{D}\langle X\rangle$ also performs better, as expected, except for a few cases where $\mathbb{C}^+\langle X\rangle$ generates more garbage collections, reducing peak memory usage at the expense of performance.

Proofs of Correctness

The goal of this section is to show that algorithms $\mathbb{D}\langle X\rangle$ and $\mathbb{C}^+\langle X\rangle$ produce the same mapping Δ for the same given trace. $\mathbb{C}\langle X\rangle$ is already known to be correct for our definition of parametric trace monitoring due to the results in [31]. As mentioned $\mathbb{C}^+\langle X\rangle$ is a straight-forward extension of $\mathbb{C}\langle X\rangle$. Thus by showing that $\mathbb{D}\langle X\rangle$ and $\mathbb{C}^+\langle X\rangle$ produce the same Δ (i.e., showing that they behave the same), we show that $\mathbb{D}\langle X\rangle$, itself, is correct for our definition of parametric monitoring.

We fix a trace $\tau = e_1e_2...e_n$, a monitor $M = (S, \mathcal{E}, \mathcal{C}, i, \sigma, \gamma)$ and a desired value set $\mathcal{G}$ in what follows. We use $\Delta_{\mathbb{C}}\langle X\rangle$ and $\Delta_{\mathbb{D}}$ to refer to the Δ in algorithms $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$, respectively. For convenience, we also let $\mathsf{timestamp} : [\mathsf{integer}\rightharpoonup\mathsf{integer}]$ be the function defined as follows: $\mathsf{timestamp}(k)$ is the value of $\mathsf{timestamp}$ in $\mathbb{D}\langle X\rangle$ at the event e_k for $0 < k \leq n$; otherwise $\mathsf{timestamp}(k)$ is undefined. $\mathsf{timestamp}$ and $\mathcal{T}$ in $\mathbb{D}\langle X\rangle$ have the following properties:

Proposition 18 *The follow holds for* $\mathsf{timestamp}$ *and* $\mathcal{T}$ *used in algorithm* $\mathbb{D}\langle X\rangle$.

1. *For* $0 < k, k' \leq n$, $k \geq k'$ *iff* $\mathsf{timestamp}(k) \geq \mathsf{timestamp}(k')$.

2. $\Delta_{\mathbb{D}}(\theta)$ *is defined iff* $\mathcal{T}(\theta)$ *is defined.*

Proof: 1. is obvious since $\mathsf{timestamp}$ is monotonic along the observed trace. 2. holds because $\Delta_{\mathbb{D}}(\theta)$ and $\mathcal{T}(\theta)$ are always defined together (lines 1 and 2 in $\mathsf{defineNew}$ and lines 6 and 7 in $\mathsf{defineTo}$). □

We next define two functions that describe *when* and *how* a monitor state is created for a parameter instance.

Definition 27 *Function* $\mathsf{set} : [[X\rightharpoonup V]\rightharpoonup\mathsf{integer}]$ *is defined as follows:* $\mathsf{set}(\theta) = k$ *if* $\Delta(\theta)$ *is initialized at* e_k. *Function* $\mathsf{MT} : [[X\rightharpoonup V]\rightharpoonup[X\rightharpoonup V]^*]$ *is defined as follows:* $\mathsf{MT}(\theta) = \theta_1...\theta_m$ *where* $\theta_m = \theta$, θ_1 *is initialized with* i, *and* $\Delta(\theta_i)$ *is initialized using* $\Delta(\theta_{i-1})$ *at some event* e *for any* $1 < i \leq m$.

Obviously, for both $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$, $\mathsf{set}(\theta)$ is defined if and only if $\mathsf{MT}(\theta)$ is defined. Let $\mathsf{set}_{\mathbb{C}}\langle X\rangle$ and $\mathsf{set}_{\mathbb{D}}$ be the set in algorithm $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$, respectively, and let $\mathsf{MT}_{\mathbb{C}}\langle X\rangle$ and $\mathsf{MT}_{\mathbb{D}}$ be the MT in algorithm $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$, respectively.

Proposition 19 *For algorithms $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$, the following hold for set and MT:*

1. *For θ_i and θ_j in $\mathsf{MT}(\theta)$, $\theta_i \sqsubset \theta_j$ if $i < j$.*
2. *If $\mathsf{MT}_{\mathbb{D}}(\theta)=\theta_1...\theta_m$ then $\mathcal{T}(\theta)=\mathsf{timestamp}(\mathsf{set}_{\mathbb{D}}(\theta_1))$.*
3. *If $\mathsf{set}_{\mathbb{D}}(\theta)$ is defined then $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta)$ is defined and $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta) \leq \mathsf{set}_{\mathbb{D}}(\theta)$.*
4. *If $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta) = \mathsf{set}_{\mathbb{D}}(\theta)$ and $\Delta_{\mathbb{C}}\langle X\rangle(\theta) = \Delta_{\mathbb{D}}(\theta)$ when they are initialized, then $\Delta_{\mathbb{C}}\langle X\rangle(\theta) = \Delta_{\mathbb{D}}(\theta)$ during the whole monitoring process.*
5. *If $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta) = \mathsf{set}_{\mathbb{D}}(\theta)$ and $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta) = \mathsf{MT}_{\mathbb{D}}(\theta)$ then $\Delta_{\mathbb{C}}\langle X\rangle(\theta) = \Delta_{\mathbb{D}}(\theta)$ during the whole monitoring process.*

Proof:

1. It follows by Definition 27 and line 6 in $\mathsf{createNewMonitorStates}$ in $\mathbb{D}\langle X\rangle$.

2. Prove by induction on the length of $\mathsf{MT}_{\mathbb{D}}(\theta)$. If $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta$, suppose that $\Delta_{\mathbb{D}}(\theta)$ is defined at event e_k, i.e., $\mathsf{set}_{\mathbb{D}}(\theta) = k$. Obviously, $\Delta_{\mathbb{D}}(\theta)$ is defined using $\mathsf{defineNew}$ in $\mathbb{D}\langle X\rangle$. Hence, $\mathcal{T}(\theta) = \mathsf{timestamp}(k)$ according to line 2 in $\mathsf{defineNew}$. Now suppose that for $0 < j$ and any θ'' s.t. $\mathsf{MT}_{\mathbb{D}}(\theta'') = \theta_1...\theta_m$ and $m < j$, $\mathcal{T}(\theta'') = \mathsf{timestamp}(\mathsf{set}_{\mathbb{D}}(\theta_1))$. If $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta_1...\theta_j$ then $\theta = \theta_j$ and $\Delta_{\mathbb{D}}(\theta)$ is defined using $\Delta_{\mathbb{D}}(\theta_{j-1})$ by Definition 27. $\mathcal{T}(\theta_j) = \mathcal{T}(\theta_{j-1})$ according to line 7 in $\mathsf{defineTo}$ in $\mathbb{D}\langle X\rangle$. By induction, $\mathcal{T}(\theta) = \mathcal{T}(\theta_{j-1}) = \mathsf{timestamp}(\mathsf{set}_{\mathbb{D}}(\theta_1))$.

3. Prove by induction on the length of $\mathsf{MT}_{\mathbb{D}}(\theta)$. We only need to show that if $\Delta_{\mathbb{D}}(\theta)$ is defined at event e_k and $\Delta_{\mathbb{C}}\langle X\rangle(\theta)$ is undefined before e_k then $\Delta_{\mathbb{C}}\langle X\rangle(\theta)$ is defined at e_k. If $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta$, suppose $\mathsf{set}_{\mathbb{D}}(\theta) = k$ and $e_k\langle\theta'\rangle$. Since θ is not initialized with another parameter instance, it should be defined using $\mathsf{defineNew}$ function in $\mathbb{D}\langle X\rangle$, which only occurs via line 4 in main. Hence, $\theta' = \theta$ and e_k is a creation event. If $\Delta_{\mathbb{C}}\langle X\rangle(\theta)$ is undefined before e_k, it will be defined at e_k because line 10 in the main function in $\mathbb{C}^+\langle X\rangle$ will be executed if $\Delta_{\mathbb{C}}\langle X\rangle(\theta)$ is undefined before line 9.

Now suppose that for any parameter instance θ'' s.t. $\mathsf{set}_{\mathbb{D}}(\theta'')$ is defined and the length of $\mathsf{MT}_{\mathbb{D}}(\theta'')$ is less than j, $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta'') \leq \mathsf{set}_{\mathbb{D}}(\theta'')$. If $\mathsf{set}_{\mathbb{D}}(\theta)$ is defined and $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta_1...\theta_j$ where $\theta_j = \theta$, let $\mathsf{set}_{\mathbb{D}}(\theta) = k$ and $e_k\langle\theta'\rangle$. By Definition 27, $\Delta_{\mathbb{D}}(\theta)$ is defined using $\Delta_{\mathbb{D}}(\theta_{j-1})$. Hence, $\mathsf{set}_{\mathbb{D}}(\theta_{j-1}) < k$ and $\theta' \sqcup \theta_{j-1} = \theta$ according to line 6 in the $\mathsf{createNewMonitorStates}$ function in $\mathbb{D}\langle X\rangle$. By induction,

$\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta_{j-1}) \leq \mathsf{set}_{\mathbb{D}}(\theta_{j-1}) < k$, that is, $\Delta_{\mathbb{C}}\langle X\rangle(\theta_{j-1})$ is defined before e_k. Therefore, if $\Delta_{\mathbb{C}}\langle X\rangle(\theta)$ is undefined before e_k, $\Delta_{\mathbb{C}}\langle X\rangle(\theta_j)$ will be defined in $\mathbb{C}^+\langle X\rangle$ at e_k because: if $\theta' = \theta$ then $\Delta_{\mathbb{C}}\langle X\rangle(\theta)$ will be defined at line 8 in main in $\mathbb{C}^+\langle X\rangle$ ($\theta_{j-1} \sqsubset \theta$ by 1.); otherwise, it will be defined at line 15 in main ($\theta' \sqcup \theta_{j-1} = \theta$).

4. In both $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$, after $\Delta(\theta)$ is defined at e_k, it will be updated using any event $e_j\langle\theta'\rangle$ with $\theta' \sqsubseteq \theta$ and $k < j$. If $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta) = \mathsf{set}_{\mathbb{D}}(\theta)$ and $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta) = \mathsf{MT}_{\mathbb{D}}(\theta)$ then $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta)$ and $\mathsf{MT}_{\mathbb{D}}(\theta)$ will be updated using the same events afterward and therefore equivalent during the whole monitoring.

5. It can be easily proved by induction on the length of $\mathsf{MT}_{\mathbb{D}}(\theta)$ and 4.

□

The following lemma shows that $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$ are equivalent for monitors that are created from the initial state.

Lemma 2 *The following hold for MT:*

1. *If $MT_{\mathbb{C}}\langle X\rangle(\theta) = \theta$ then $MT_{\mathbb{D}}(\theta) = \theta$ and $set_{\mathbb{C}}\langle X\rangle(\theta) = set_{\mathbb{D}}(\theta)$.*
2. *If $MT_{\mathbb{D}}(\theta) = \theta$ then $MT_{\mathbb{C}}\langle X\rangle(\theta) = \theta$ and $set_{\mathbb{C}}\langle X\rangle(\theta) = set_{\mathbb{D}}(\theta)$.*

Proof:

1. If $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta) = \theta$, suppose that $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta) = k$. Obviously, $\Delta_{\mathbb{C}}\langle X\rangle(\theta)$ is defined by the defineNew function in $\mathbb{C}^+\langle X\rangle$, which only occurs when e_k is a creation event and comes with the parameter instance θ. Also, for all $\theta' \sqsubset \theta$, $\Delta_{\mathbb{C}}\langle X\rangle(\theta')$ is undefined before e_k; otherwise, $\Delta_{\mathbb{C}}\langle X\rangle(\theta)$ should be defined using $\Delta_{\mathbb{C}}\langle X\rangle(\theta')$ at line 8 in main in $\mathbb{C}^+\langle X\rangle$. By Proposition 19 3., $\Delta_{\mathbb{D}}(\theta)$ and $\Delta_{\mathbb{D}}(\theta')$, for all $\theta' \sqsubset \theta$, are undefined before e_k. So $\Delta_{\mathbb{D}}(\theta)$ cannot be defined in the createNewMonitorStates function in $\mathbb{D}\langle X\rangle$ using some $\theta' \sqsubset \theta$ when e_k is encountered. Hence, the condition at line 3 in main in $\mathbb{D}\langle X\rangle$ is satisfied and line 4 will be executed to initialize $\Delta_{\mathbb{D}}(\theta)$ using defineNew in $\mathbb{D}\langle X\rangle$. Therefore, $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta$ and $\mathsf{set}_{\mathbb{D}}(\theta) = k = \mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta)$.

2. By Proposition 19.3., if $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta$ and $\mathsf{set}_{\mathbb{D}}(\theta) = k$ then $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta)$ is defined before or at e_k. Assume that $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta) = \theta_1..\theta_m$ and $m > 1$. Then we have 1) $\theta_1 \sqsubset \theta$ by Proposition 19 1.; 2) $\mathsf{MT}_{\mathbb{D}}(\theta_1) = \mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta_1) = \theta_1$ and $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta_1) = \mathsf{set}_{\mathbb{D}}(\theta_1)$ 1.; 3) $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta_1) < \mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta) \leq \mathsf{set}_{\mathbb{D}}(\theta)$ by Proposition 19.3. Let $e_k\langle\theta'\rangle$. Since $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta$, $\Delta_{\mathbb{D}}(\theta)$ is defined using defineNew via line 4 in main in $\mathbb{D}\langle X\rangle$ when e_k is encountered. Hence, $\theta = \theta'$. However, since $\Delta_{\mathbb{D}}(\theta_1)$ is defined before e_k, the condition at line 2 in defineNew is satisfied and $\Delta_{\mathbb{D}}(\theta)$ cannot be defined at e_k. Contradiction reached. Therefore, $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta) = \theta$. By 1., $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta) = \mathsf{set}_{\mathbb{D}}(\theta)$. □

Proposition 20 *For algorithms $\mathbb{C}^+\langle X\rangle$ and $\mathbb{D}\langle X\rangle$, the following hold:*

1. *If $MT_{\mathbb{C}}\langle X\rangle(\theta) = MT_{\mathbb{D}}(\theta)$ then for any $\theta' \in MT_{\mathbb{C}}\langle X\rangle(\theta)$, $set_{\mathbb{C}}\langle X\rangle(\theta') = set_{\mathbb{D}}(\theta')$.*

2. *If $MT_{\mathbb{C}}\langle X\rangle(\theta) = MT_{\mathbb{D}}(\theta)$ then $\Delta_{\mathbb{C}}\langle X\rangle(\theta) = \Delta_{\mathbb{D}}(\theta)$ during the whole monitoring.*

Proof:

1. Suppose $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta) = \theta_1, .., \theta_m$. Prove by induction on $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta)$. For θ_1, since $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta_1) = \theta_1$, $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta_1) = \mathsf{set}_{\mathbb{D}}(\theta_1)$ by Lemma 2.1. Now suppose that for some $1 < jleqm$, $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta_i) = \mathsf{set}_{\mathbb{D}}(\theta_i)$ for any $0 < i < j$. Assume that $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta_j) \neq \mathsf{set}_{\mathbb{D}}(\theta_j)$. We have $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta_j) < \mathsf{set}_{\mathbb{D}}(\theta_j)$ by Proposition 19.3. Let $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta_j) = k$ and $e_k\langle\theta''\rangle$. Since $\theta'' \sqcup \theta_{j-1} = \theta_j$, we have $\theta'' \not\sqsubset \theta_{j-1}$. Also, $\mathsf{disable}(\theta'') > \mathsf{timestamp}(k) > \mathcal{T}(\theta_{j-1})$ after e_k. Let $\mathsf{set}_{\mathbb{D}}(\theta) = g$. We have that $\Delta_{\mathbb{D}}(\theta_j)$ cannot be defined at e_g using $\Delta_{\mathbb{D}}(\theta_{j-1})$ because $g > k$ and θ'' will satisfy the condition at line 2 in $\mathsf{defineTo}$ in $\mathbb{D}\langle X\rangle$. Contradiction found. Therefore, $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta_j) = \mathsf{set}_{\mathbb{D}}(\theta_j)$.

2. Follow by 1. and Proposition 19.5. □

Let $\Delta^{\tau}_{\mathbb{C}}\langle X\rangle$ be the Δ after $\mathbb{C}^+\langle X\rangle$ processes τ and $\Delta^{\tau}_{\mathbb{D}}$ be the Δ after $\mathbb{D}\langle X\rangle$ processes τ.

Proposition 21 *The following holds:*

1. *If $\gamma(\Delta^{\tau}_{\mathbb{C}}\langle X\rangle(\theta)) \in \mathcal{G}$ and for any $\theta_i \in MT_{\mathbb{C}}\langle X\rangle(\theta)$, $i > 1$, let $set_{\mathbb{C}}\langle X\rangle(\theta_i) = k$, we have $Dom(\theta_{i-1}) \in enable^X_{\mathcal{G}}(e_k)$.*

2. *If $\gamma(\Delta^{\tau}_{\mathbb{C}}\langle X\rangle(\theta)) \in \mathcal{G}$ then $MT_{\mathbb{C}}\langle X\rangle(\theta) = MT_{\mathbb{D}}(\theta)$.*

Proof:

1. Suppose that the sliced trace for θ is $\tau_\theta = e'_1\langle\theta'_1\rangle...e'_h\langle\theta'_h\rangle$. Then $\sigma(\tau_\theta) = \Delta^{\tau}_{C}(\theta)$, according to Theorem 3 in [31]. Since $\gamma(\Delta^{\tau}_{\mathbb{C}}\langle X\rangle(\theta)) \in \mathcal{G}$, $P(\tau_\theta) \in \mathcal{G}$. Also, since $\Delta_{\mathbb{C}}\langle X\rangle(\theta_i)$ is defined at e_k, $e_k \in \tau_\theta$ and it is the first occurrence of e_k in τ_θ. Suppose that e'_n is the first occurrence of e_k in τ_θ. Then $\mathsf{enable}_\tau(e_k) = \{e'_1, ..., e'_{n-1}\}$ by Definition 24. For any $0 < j < n$, let $e'_j\langle\theta''\rangle$, then $\theta'' \sqsubseteq \theta_{i-1}$; otherwise, e'_j should not be contained in the slice for θ_{i-1} and thus not in the slice for θ_i (since $\Delta_{\mathbb{C}}\langle X\rangle(\theta_i)$ is initialized using $\Delta_{\mathbb{C}}\langle X\rangle(\theta_{i-1})$.) Hence, $\cup^X_{\{e'_1,...,e'_{n-1}\}} = Dom(\theta_{i-1})$, that is, $Dom(\theta_{i-1}) \in \mathsf{enable}^X_{\mathcal{G}}(e_k)$ by Definition 26.

2. Suppose that $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta) = \theta_1, ..., \theta_m$. Prove by induction on $\mathsf{MT}_C(\theta)$. For θ_1, $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta_1) = \theta_1$. Hence, $\mathsf{MT}_{\mathbb{D}}(\theta_1) = \theta_1$ by Lemma 2. Now suppose that for some $1 < j \leq m$, we have $\mathsf{MT}_{\mathbb{D}}(\theta_{j-1}) = \mathsf{MT}_C(\theta_{j-1}) = \theta_1, ..., \theta_{j-1}$. Let $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta_j) = k$ and $e_k\langle\theta'\rangle$. By Proposition 19 3., $\Delta_{\mathbb{D}}(\theta_j)$ is undefined before e_k. Also, $\theta' \sqcup \theta_{j-1} = \theta_j$ due to line 15 in main in $\mathbb{C}^+\langle X\rangle$.

By 1., $Dom(\theta_{j-1}) \in \mathsf{enable}^X_{\mathcal{G}}(e_k)$. Hence, $\Delta_{\mathbb{D}}(\theta_j)$ will be defined at e_k because of the loop from line 4 - 8 in createNewMonitorStates in $\mathbb{D}\langle X\rangle$. We only need to show that $\Delta_{\mathbb{D}}(\theta_j)$ is defined using $\Delta_{\mathbb{D}}(\theta_{j-1})$. Assume that $\Delta_{\mathbb{D}}(\theta_j)$ is defined using $\Delta_{\mathbb{D}}(\theta'')$ and $\theta'' \neq \theta_{j-1}$. Then we have $\theta'' \sqcup \theta' = \theta_j$. $\theta'' \not\sqsubset \theta_{j-1}$ because the loop from line 1 to line 10 in createNewMonitorStates in $\mathbb{D}\langle X\rangle$ is carried out in a reverse topological order. Also, $\theta_{j-1} \not\sqsubset \theta''$ because the loops from line 2 to line 6 and from line 12 to line 18 in main in $\mathbb{C}^+\langle X\rangle$ are carried out in a reverse topological order. Such situation, i.e., θ_j does not have a maximum sub-instance, is impossible according to the proof for algorithm $\mathbb{A}\langle X\rangle$ in [31]. Contradiction found. Therefore, $\Delta_{\mathbb{D}}(\theta_j)$ is defined using $\Delta_{\mathbb{D}}(\theta_{j-1})$ at e_k. We then have $\mathsf{MT}_{\mathbb{D}}(\theta_j) = \mathsf{MT}_{\mathbb{D}}(\theta_{j-1})\theta_j = \mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta_{j-1})\theta_j = \mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta_j)$. By induction, $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta_m) = \mathsf{MT}_{\mathbb{D}}(\theta_m)$.

□

Proposition 22 *If $\Delta^{\tau}_{\mathbb{D}}(\theta)$ is defined then $MT_{\mathbb{C}}\langle X\rangle(\theta) = MT_{\mathbb{D}}(\theta)$.*

Proof: Suppose that $\mathsf{MT}_{\mathbb{D}}(\theta) = \theta_1, ..., \theta_m$. Prove by induction on $\mathsf{MT}_{\mathbb{D}}(\theta)$. For θ_1, $\mathsf{MT}_{\mathbb{D}}(\theta_1) = \theta_1$. Hence, $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta_1) = \theta_1$ by Lemma 2.2. Now suppose that for some $1 < j \leq m$, we have $\mathsf{MT}_{\mathbb{D}}(\theta_{j-1}) = \mathsf{MT}_{C}(\theta_{j-1}) = \theta_1, ..., \theta_{j-1}$. Let $\mathsf{set}_{\mathbb{D}}(\theta_j) = k$ and $e_k\langle\theta'\rangle$.

Suppose that $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta_j) = \theta^j_1 ... \theta^j_h$ where $\theta^j_h = \theta_j$. We first show that $\theta_1 = \theta^j_1$ by contradiction. Assume $\theta_1 \neq \theta^j_1$. Let $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta^j_1) = p^j$ and $\mathsf{set}_{\mathbb{D}}(\theta_1) = p$. Since $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta^j_1) = \theta^j_1$ and $\mathsf{MT}_{\mathbb{D}}(\theta_1) = \theta_1$, we have that $e_{p^j}\langle\theta^j_1\rangle$, $e_p\langle\theta_1\rangle$ and they are both creation events. We also have $\mathcal{T}_{\mathbb{D}}(\theta_1) = \mathsf{timestamp}(p)$. By Proposition 19.2, $\Delta_{\mathbb{D}}(\theta^j_1)$ is not defined before p^j. Hence, $\Delta_{\mathbb{D}}(\theta^j_1)$ is defined at p^j and $\mathcal{T}_{\mathbb{D}}(\theta^j_1) = \mathsf{timestamp}(p^j)$. Also, $\mathsf{disable}(\theta^j_1) > \mathcal{T}_{\mathbb{D}}(\theta^j_1)$ since line 6 in main of algorithm $\mathbb{D}\langle X\rangle$ is executed after $\mathcal{T}_{\mathbb{D}}(\theta^j_1)$ is defined at line 4. Since $\theta_1 \neq \theta^j_1$, $p^j \neq p$; in other words, either $p^j < p$ or $p^j > p$. Therefore, either $\mathcal{T}_{\mathbb{D}}(\theta^j_1) < \mathcal{T}_{\mathbb{D}}(\theta_1)$ or $\mathcal{T}_{\mathbb{D}}(\theta_1) < \mathcal{T}_{\mathbb{D}}(\theta^j_1) < \mathsf{disable}(\theta^j_1)$ by Proposition 18.1. Let θ_n be the first parameter instance in $\mathsf{MT}_{\mathbb{D}}(\theta_j)$ s.t. $\theta^j_1 \sqsubset \theta_n$ and $\theta^j_1 \not\sqsubset \theta_{n-1}$, $n > 1$, and let $\mathsf{set}_{\mathbb{D}}(\theta_n) = p_n$. Then $\Delta_{\mathbb{D}}(\theta_n)$ is defined in the defineTo function in $\mathbb{D}\langle X\rangle$ at e_{p_n} using $\Delta_{\mathbb{D}}(\theta_{n-1})$. However, it is impossible since θ^j_1 satisfies the condition at line 2 in defineTo and prevents defining $\Delta_{\mathbb{D}}(\theta_n)$ at e_{p_n}. Contradiction found and $\theta_1 = \theta^j_1$.

Assume that $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta_j) \neq \mathsf{MT}_{\mathbb{D}}(\theta_j)$. We can find $l > 1$ s.t. $\theta^j_l \neq \theta_l$ and $\theta^j_i = \theta_i$ for any $0 < i < l$. Let $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta^j_l) = k$ and $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta_l) = g$. Suppose $e_{n_l}\langle\theta''\rangle$. We have $\theta^j_{l-1} \sqcup \theta'' = \theta^j_l$; so $\theta'' \not\sqsubset \theta^j_{l-1}$. Also, $\mathsf{disable}(\theta'') > \mathcal{T}(\theta^j_l) = \mathcal{T}(\theta^j_1) = \mathcal{T}(\theta_1)$ after e_k. $k < g$ is impossible; otherwise, $\Delta_{\mathbb{D}}(\theta_l)$ cannot be defined at e_g using $\Delta_{\mathbb{D}}(\theta_{l-1})$ because θ'' will satisfy the condition at line 2 in defineTo in $\mathbb{D}\langle X\rangle$. Hence, $k > g \geq \mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta_l)$ by Proposition 19.3. In other words, $\Delta_{\mathbb{C}}\langle X\rangle(\theta_l)$ is

defined before e_k. Therefore, $\theta_l \notin \mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta_j)$ but $\theta_l \subseteq \theta_j$. Then we can find $\theta_p^j \in \mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta_j)$ s.t. $\theta_l \sqsubset \theta_p^j$ and $\theta_l \not\sqsubset \theta_i$ for any $0 < i < p$. However, suppose $\mathsf{set}_{\mathbb{C}}\langle X\rangle(\theta_p^j) = n$, then at event e_n, we have $\theta_l \sqsubset \theta_p^j$ and $\theta_l \not\sqsubset \theta_{p-1}^j$. According to the proof for algorithm $\mathbb{A}\langle X\rangle$ in [31], we should have $\theta_{p-1}^j \sqsubset \theta_l$, which means that $\Delta_{\mathbb{C}}\langle X\rangle(\theta_p^j)$ should be defined using $\Delta_{\mathbb{C}}\langle X\rangle(\theta_l)$. Contradiction found. Therefore, $\mathsf{MT}_{\mathbb{C}}\langle X\rangle(\theta_j) = \mathsf{MT}_{\mathbb{D}}(\theta_j)$.

□

Theorem 4 *The following holds:*

1. *if* $\gamma(\Delta_{\mathbb{C}}^{\tau}\langle X\rangle(\theta)) \in \mathcal{G}$ *then* $\gamma(\Delta_{\mathbb{D}}^{\tau}(\theta)) = \gamma(\Delta_{\mathbb{C}}^{\tau}\langle X\rangle(\theta))$;
2. *if* $\gamma(\Delta_{\mathbb{D}}^{\tau}(\theta)) \in \mathcal{G}$ *then* $\gamma(\Delta_{\mathbb{C}}^{\tau}\langle X\rangle(\theta)) = \gamma(\Delta_{\mathbb{D}}^{\tau}(\theta))$;
3. $\gamma(\Delta_{\mathbb{C}}^{\tau}\langle X\rangle(\theta)) \in \mathcal{G}$ *iff* $\gamma(\Delta_{\mathbb{D}}^{\tau}(\theta)) = \gamma(\Delta_{\mathbb{C}}^{\tau}\langle X\rangle(\theta))$ *iff* $\gamma(\Delta_{\mathbb{D}}^{\tau}(\theta)) \in \mathcal{G}$.

Proof:

1. By Proposition 21 and Proposition 20.2, $\Delta_{\mathbb{D}}^{\tau}(\theta) = \Delta_{\mathbb{C}}^{\tau}\langle X\rangle(\theta)$. Hence, $\gamma(\Delta_{\mathbb{D}}^{\tau}(\theta)) = \gamma(\Delta_{\mathbb{C}}^{\tau}\langle X\rangle(\theta))$.

2. Follow by Proposition 22 and Proposition 20.2.

3. Follow by 1 and 2. □

Theorem 4 states that a trace slice for θ is reported by $\mathbb{C}^{+}\langle X\rangle$ to be in $\mathcal{G}$ if and only if it is also reported by $\mathbb{D}\langle X\rangle$ to be in $\mathcal{G}$. In other words,$\mathbb{C}^{+}\langle X\rangle$ and $\mathbb{D}\langle X\rangle$ are equivalent for those parameter instances whose trace slices are in $\mathcal{G}$. Thus $\mathbb{D}\langle X\rangle$ is complete and sound.

Chapter 5

JavaMOP

JavaMOP is a language instance of the MOP framework for Java. It provides several interfaces, including a web-based interface, a command-line interface and an Eclipse-based GUI, providing the developer with different means to manage and process MOP specifications. The JavaMOP implementation follows the server-clicient architecture of MOP, as shown in Figure 3.2, to flexibly support these various interfaces, as well as for portability reasons. AspectJ [9] is employed for monitor integration: JavaMOP translates the given JavaMOP specification into AspectJ code, which is then merged within the original program by the AspectJ compiler. All the logic plugins implemented the MOP framework are supported in JavaMOP.

One might expect some loss of efficiency for MOP's genericity of logics. However, the JavaMOP-generated monitors can yield very reasonable runtime overhead in practice, even for properties requiring intensive runtime checking: on the order of 10% or lower, and as efficient as the hand optimized monitoring code in most cases.

5.1 JavaMOP Specifications

Figure 5.1 shows the syntax of JavaMOP Specifications. The syntax is defined in BackusCNaur Form (BNF) [19] extended with $\{p\}$ for zero or more and $[p]$ for zero or one repetitions of p. We next give an itemized explanation of the syntactic definitions and also two examples for better understanding. JavaMOP specification syntax is defined on the Java syntax and the AspectJ syntax; one can refer to [53] for the former and [10] for the latter.

⟨**Specification**⟩**.** Every JavaMOP specification is composed of the following components: a list of modifiers, a name for the specification, a list of parameters which can be empty, a list of declarations that define internal variables of the generated monitor, a list of event definitions, and a list of property specifications. All the components are optional except for the name and parameters of the specification.

⟨Specification⟩	::=	{⟨Modifier⟩} ⟨Id⟩ ⟨Parameters⟩ "{" {⟨Declaration⟩} {⟨Event⟩} { ⟨Property⟩ {⟨Property Handler⟩}} "}"
⟨Modifier⟩	::=	"unsynchronized" | "decentralized" | "perthread" | "suffix"
⟨Event⟩	::=	"event" ⟨Id⟩ ⟨Event Definition⟩ ⟨Action⟩
⟨Property⟩	::=	⟨Logic Name⟩ " : " ⟨Logic Syntax⟩
⟨Property Handler⟩	::=	"@"⟨Logic State⟩ ⟨Action⟩
⟨Event Definition⟩	::=	⟨Advice Specification⟩ " : " ⟨Extended Pointcut⟩
⟨Action⟩	::=	"{" [⟨Statements⟩] "}"
⟨Extended Pointcut⟩	::=	⟨Pointcut⟩ | ⟨Extended Pointcut⟩ "&&" ⟨Extended Pointcut⟩ | "thread" "(" ⟨Id⟩ ")" | "condition" "(" ⟨Boolean Expression⟩ ")"
⟨Parameters⟩	::=	"(" [⟨Parameter⟩{ "," ⟨Parameter⟩}] ")"
⟨Parameter⟩	::=	⟨Type Pattern⟩ ⟨Id⟩
⟨Type Pattern⟩	::=	⟨! − −AspectJ Type Pattern − −⟩
⟨Id⟩	::=	⟨! − −Java Identifier − −⟩
⟨Declaration⟩	::=	⟨! − −Java variable declaration − −⟩
⟨Statements⟩	::=	⟨! − −Java statements − −⟩
⟨Boolean Expression⟩	::=	⟨! − −Java boolean expressions − −⟩
⟨Advice Specification⟩	::=	⟨! − −AspectJ AdviceSpec − −⟩
⟨Pointcut⟩	::=	⟨! − −AspectJ Pointcut − −⟩
⟨Logic Name⟩	::=	⟨! − −Name of the used logic − −⟩
⟨Logic Syntax⟩	::=	⟨! − −Property specified in the used logic − −⟩
⟨Logic State⟩	::=	⟨! − −State of the generated monitor − −⟩

Figure 5.1: JavaMOP Specification Syntax in BNF

⟨Modifier⟩. JavaMOP supports four modifiers which can be used to configure the running mode of the monitor. As the grammar shows, modifiers are placed at the beginning of the specification definition. Presently, four modifiers are supported by JavaMOP:

- "unsynchronized": When this modifier is specified, the monitor state is not protected against concurrent accesses during monitoring; otherwise, the accesses to the monitor state will be synchronized. The unsynchronized monitor is faster, but may suffer from races on its state updates, if the monitored program has multiple threads.

- "decentralized": When this modifier is specified, decentralized monitor indexing is used to store the monitor for different parameter instances. If it is not specified, the default mode is centralized monitor indexing. Decentralized indexing means that the indexing trees used to search for monitors are scattered all over the running system as additional fields of objects of interest, while centralized indexing means the indexing trees are stored in a common place. Decentralized indexing typically yields lower runtime overhead, but it does not work for all

settings, such as when the objects of interest cannot be modified with aspects. Section 5.2 explains how centralized and decentralized indexing work.

- "perthread": When this modifier is specified, each thread is monitored separately, and every monitor never receives events from more than one thread. JavaMOP will optimize the monitoring code accordingly. For example, perthread monitors are automatically unsynchronized.
- "suffix": When this modifier is specified, suffix matching is used. If it is not specified, the default mode is total matching. In total matching, the given property is checked against the whole execution trace. In suffix matching, the given property is checked against every suffix of the execution trace (see Section 3.2). For example, for a regular pattern `a b` and a trace `a a b`, total matching will find no match of the pattern, while suffix matching will generate one match for the suffix `a b` starting from the second `a`.

⟨**Event**⟩**.** The event declaration defines events that will be observed at runtime and referred to in the specified property (see below). Every event declaration begins with keyword `event` followed by an ⟨Id⟩ that gives the name of the event. ⟨Event Definition⟩ and ⟨Action⟩ define a condition and its behavior when triggered.

⟨**Property**⟩**.** Every JavaMOP specification may contain zero or more properties. A property consists of a named formalism (⟨Logic Name⟩), followed by a colon, followed by a property specification using the named formalism (⟨Logic Syntax⟩) and usually referring to the declared events. JavaMOP, like all MOP instances, is not bound to any particular property specification formalism. New formalisms can be added to a JavaMOP installation by means of logic plugins (see Section 3.3).

A JavaMOP specification containing no property specification is called raw. Raw specifications are useful when no existing logic plugin is powerful or efficient enough to specify the desired property; in that case, one embeds the custom monitoring code manually within the event action.

⟨**Property Handler**⟩**.** Property handlers can be defined for certain states (those states are designated by ⟨Logic State⟩) of the generated monitor. A property handler consists of any arbitrary Java statements that will be invoked whenever the designated state is reached in the generated monitor. Just as the event action, the handler may modify the program or the monitor state. The monitor states to which one can associate handlers are determined by the underlying formalism, e.g., validation or violation in linear temporal logic specifications, match or fail for ERE or CFG, or a particular state in a finite state machine description.

⟨**Event Definition**⟩**.** ⟨Event Definition⟩ in JavaMOP makes uses of AspectJ syntax. It consists of ⟨Advice Specification⟩ and ⟨Extended Pointcut⟩, which define where the event is triggered by the program.

⟨**Action**⟩**.** Events and handlers can also have arbitrary code associated with them, called an action. The action is run when the associated event is observed, or the handler triggered. An action is surrounded by curly braces ("" and ""), and can contain any arbitrary Java statements. ⟨Action⟩s may modify the program or the monitor state. They have access to the special expressions defined in the ⟨Statements⟩.

⟨**Extended Pointcut**⟩**.** ⟨Extended Pointcut⟩ extends the AspectJ Pointcut with the following two JavaMOP-specific pointcuts:

- thread captures the current thread and takes an identifier as the parameter. The identifier is bound to the running thread object. The captured thread object is stored into the given variable so that the monitor can access it later.
- condition takes a boolean expression as the parameter. The event is triggered only when the given boolean expression evaluates to true.

⟨**Parameters**⟩**,** ⟨**Parameter**⟩**, and** ⟨**Type Pattern**⟩**.** ⟨Parameters⟩ is a comma-separated list of parameters. ⟨Parameter⟩ is composed of a type pattern (⟨Type Pattern⟩) that defines the type of the parameter and an identifier (⟨Id⟩) that gives the name of the parameter. *TypePattern* is the AspectJ type pattern, which allows for using wildcards, such as + and *, to define patterns of types. The list of parameters defines which parameters may be used in the events of the specification.

⟨**Id**⟩**,** ⟨**Declaration**⟩**,** ⟨**Statements**⟩**, and** ⟨**Boolean Expression**⟩**.** ⟨Id⟩ and ⟨Declaration⟩ are the ordinary Java identifier and the Java declaration, respectively. ⟨Statements⟩ are ordinary Java statements. ⟨Statements⟩ also support three special expressions that can be used in the property handler:

- __RESET: a special expression (evaluates to void) that resets the monitor to its initial state;
- __LOC: a string variable that evaluates to the line number where the current event is generated;
- __MONITOR: a special variable that evaluates to the current monitor object, so that one can read/write monitor variables.

BExp is the ordinary Java boolean expression.

⟨Advice Specification⟩ and ⟨Pointcut⟩. ⟨Advice Specification⟩ and ⟨Pointcut⟩ refer to the AspectJ AdviceSpec and Pointcut, respectively. JavaMOP borrows AspectJ syntax for its expressiveness and also to facilitate code synthesis.

⟨Logic Name⟩, ⟨Logic Syntax⟩, and ⟨Logic State⟩. ⟨Logic Name⟩ designates the logic to specify the desired property. The current version of JavaMOP provides the following logic names that one can use in the property specification:

- FSM: Finite State Machines
- ERE: Extended Regular Expressions
- CFG: Context Free Grammars
- PTLTL: Past Time Linear Temporal Logic
- FTLTL: Future Time Linear Temporal Logic
- PTCARET: Past Time LTL with Calls and Returns

⟨Logic Syntax⟩ is the property specified using the named logic and ⟨Logic State⟩ chooses certain states of the generated monitor with which one can associate an action. They both vary from one logic to another and the interested reader may refer to [26] for more explanation of each supported logic.

Examples

In addition to the example in Section 2.1, we next show two examples to illustrate JavaMOP specifications. The first one, shown in Figure 5.2, is a multi-formulae specification that expands the UnsafeMapIter example discussed at the beginning of Chapter 4. More precisely, Figure 5.2 shows a JavaMOP specification of the UnsafeMapIter property using five different formalisms: finite state machines (FSM), extended regular expressions (ERE), context-free grammars (CFG), future-time linear temporal logic (FTLTL), and past-time linear temporal logic (PTLTL). Because each of the properties in Figure 5.2 is the same, five messages will be reported whenever an Iterator is incorrectly used after an update to the underlying Map. We show all five of them to emphasize the formalism-independence of our approach. On the first line, we name the specified property and give the

```
UnsafeMapIterator(Map m, Collection c, Iterator i){
   event create_coll after(Map m) returning(Collection c) : (call(* Map.values()) || call(* Map.keySet())) && target(m) {}
   event create_iter after(Collection c) returning(Iterator i) : call(* Collection+.iterator()) && target(c) {}
   event use_iter before(Iterator i) : call(* Iterator+.next()) && target(i) {}
   event update_map after(Map m) : (call(* Map.remove*(..)) || call(* Map.put*(..))
      || call(* Map.putAll*(..)) || call(* Map.clear())) && target(m) {}

   fsm: start [ create_coll -> s1 ]
        s1    [ update_map -> s1, create_iter -> s2 ]
        s2    [ use_iter -> s2,   update_map -> s3  ]
        s3    [ update_map -> s3, use_iter -> end   ]
        end   []
      @end{ System.out.println("fsm: Accessed Invalid Iterator!"); __RESET; }

   ere : create_coll update_map* create_iter use_iter* update_map update_map* use_iter
      @match{ System.out.println("ere: Accessed Invalid Iterator!"); __RESET; }

   cfg : S -> create_coll Updates create_iter Nexts update_map Updates use_iter,
         Nexts -> Nexts use_iter | epsilon,
         Updates -> Updates update_map | epsilon
      @match{ System.out.println("cfg: Accessed Invalid Iterator!"); __RESET;}

   ftltl: <>(create_coll /\ <> (create_iter /\ <> (update_map /\ <> use_iter)))
      @validation{ System.out.println("ftltl: Accessed Invalid Iterator!"); __RESET; }

   ptltl: use_iter -> ((<*> (create_iter /\ (<*> create_coll))) -> ((!update_map) Since create_iter))
      @violation{ System.out.println("ptltl: Accessed Invalid Iterator!"); __RESET; }
 }
```

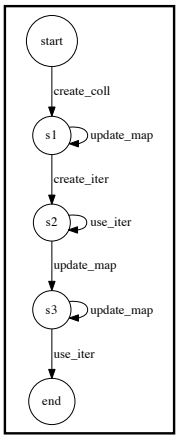

Figure 5.2: FSM, ERE, CFG, FTLTL, and PTLTL UnsafeMapIter. Inset: Graphical Depiction of the Property.

parameters used in the specification. Then we define the involved events using the AspectJ syntax. For example, create_coll is defined as the return value of functions values and keyset of Map. We adopt AspectJ syntax to define events in JavaMOP because it is an expressive language for defining observation points in a Java program. As mentioned, every event may instantiate some parameters at runtime. This can be seen in Figure 5.2: create_coll will instantiate parameters m and c using the target and the return value of the method call.

Figure 5.3 shows a raw MOP specification that detects SQL-injection attacks [4]: malicious users try to corrupt a database by inserting unsafe SQL statements into the input to the system.

In SQL injection, a string is "tainted" when it depends upon some user input; when a tainted string is used as a SQL query, it should be checked to avoid potential attacks. In Figure 5.3, a set is declared to store all tainted strings in the generated monitor. Three types of events need to be monitored: userInput occurs when a string is obtained from user input (by calling ServletRequest.getParameter()); propagate occurs when a new string is created from another string; finally, usage occurs at using a string as a query.

Appropriate actions are triggered at observed events: at userInput, the user input string is added to the tainted set; at propagate, if the new string is created from a tainted string then it is marked as tainted, too; at usage, if the query string is tainted then a provided method, called Util.checkSafeQuery, is called to check the safety of the query. Thus the safety check, which can be an expensive operation, is invoked dynamically, on a by-need basis. In particular, for efficiency and separation of concerns

```
SQLInjection () {
Set taintedStrings = new IdentitySet();
event userInput after() returning (String tainted):
   call(String ServletRequest.getParameter(..))
   { taintedStrings.put(tainted); }
event propagate after(String s) returning (StringBuffer newS) :
   call(StringBuffer StringBuffer.new(..)) && args(s)
|| call(StringBuffer StringBuffer.append(..)) && args(s)
   ...
   { if (taintedStrings.contains(s)) taintedStrings.put(newS.toString()); }
event usage before(String s) : call(* Statement.executeQuery(..)) && args(s)
   { if taintedStrings.contains(s) Util.checkSafeQuery(s); }
}
```

Figure 5.3: Raw MOP specification for SQL injection

reasons, a developer may even ignore the SQL injection safety aspect when writing code; the raw MOP specification above will take care of this aspect entirely.

This example shows that the event/action abstraction provided by raw MOP specifications is easy to master and useful for defining interesting safety properties compactly and efficiently. Event names were not needed here, so they could have been omitted. No property specifications are needed in raw MOP specifications; the developer fully implements the monitoring process by providing event actions using the target programming language.

5.2 Implementation

As mentioned in Section 3.3, JavaMOP mainly implements the Java language client of the MOP framework. We next discuss the implementation of suffix matching and parametric monitoring in JavaMOP in more details.

5.2.1 Suffix Matching

Based on the discussion of suffix matching in Section 3.2, we have implemented a logic-independent extension of JavaMOP to also support suffix matching. As discussed, although total matching and suffix matching have inherently different semantics, it is not difficult to support suffix matching in a total matching setting, if one maintains *a set of monitor states* during monitoring and *creates a new monitor instance at each event* (this amounts to checking the property on each suffix incrementally). However, the situation becomes more complicated when one wants to develop a logic-independent

solution, since different logical formalisms can have different state representations. For example, the monitor state can be an integer when the monitor is based on a state machine, a vector like the past-time LTL monitor, or a stack such as the CFG monitor discussed below. Hence, our solution is to *treat every monitor as a blackbox* without assumptions on its internal state. Also, instead of maintaining a set of monitor states in the monitor, we use a wrapper monitor that keeps a set of total matching monitors as its state for suffix matching. For simplicity, from now on, when we say "monitor" without specific constraints, we mean the monitor generated for total matching. When an event is received, the wrapper monitor for suffix matching operates as follows:

1. create a new monitor and add it to the "suffix matching" monitor set;
2. invoke every monitor in the monitor set to handle the received event;
3. if a monitor enters its "pattern fail" state, remove it from the monitor set;
4. if a monitor enters its "pattern match" state, report the pattern match.

The third step is used to keep the "suffix matching" monitor set small by removing unnecessary monitors. Informally, this implements suffix matching semantics because each total monitor is monitoring a suffix of the current trace and "pattern match" is only reported if one of the suffixes is valid.

Using our current implementation of suffix matching in JavaMOP, one may further improve the monitoring efficiency if the monitor provides an optional interface, namely, an equals method that compares two monitors with regard to their internal states, and a hashCode method used to reduce the amount of calls to equals. This interface is used to populate a Java HashSet: the combination of the definition of hashCode and equals ensures the monitors in the HashSet are declared duplicates, and removed, based on monitor state rather than memory location. This interface can be easily generated by each JavaMOP logic plugin, because it has full knowledge of the monitor semantics. It is important to note that our approach does *not depend on the underlying specification formalism.*

Moreover, JavaMOP already requires the logic plugin to designate *creation events* that are the starting events of a validating trace, in order to avoid the overhead of unnecessary monitor creation. A new monitor instance need be created only at creation events. This feature is especially useful when combined with suffix matching, which requires creating a new monitor at every event, if no creation events are chosen.

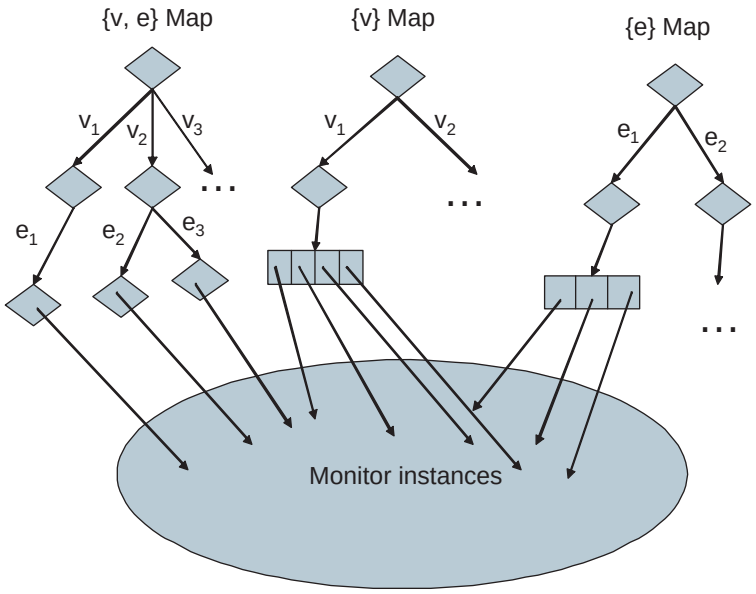

Figure 5.4: Centralized indexing for MOP spec in Figure 2.1

5.2.2 Parametric Monitoring

JavaMOP implements the logic-independent parametric monitoring framework instroduced in Chapter 4, which allows one to write parametric specifications in JavaMOP using any of the existing logic-plugins in MOP. One would expect that such a genericity must come at a performance price. However, as shown in Section 7.2, our generic technique presented next produces significantly less runtime overhead than other existing RV systems, including Tracematches and PQL.

As shown in algorithm $\mathbb{D}\langle X\rangle$ in Figure 4.16, a monitor instance checking the specified property will be created for every specific group of values of parameters; if a monitor instance `m` is created for a group of values containing `o`, then we say that `m` is *related* to `o`. For the `UnsafeEnum` specification in Figure 2.1, a monitor instance will be created for every pair of concrete `v` and `e` if `e` is the enumeration of `v`. When a relevant event occurs, concrete values are bound to the event parameters and used to look up related monitor instances; related monitors are then invoked to handle the observed event. Several monitors can be triggered by an event since the event may contain fewer parameters than the parameters of the enclosing specification. For the `UnsafeEnum` example, when an `updatesource` event occurs, the target `Vector` object is bound to the parameter `v` and used to find all the related monitors to process `updatesource` (there may be several enumerations of `v`).

The monitor lookup process is external to the monitor in our approach and makes no assumption on the implementation of the monitor; consequently, it is independent of the monitor generation

algorithm. Also, the monitor does not need to be aware of the parameter information and can proceed solely according to the observed event. Hence, the monitoring process for parametric specifications is divided into two parts in MOP: the logic-specific monitor (generated by the logic plugin) and the logic-independent lookup process (synthesized by the specification processor).

Current runtime verification approaches supporting logics with universal quantifiers construct a centralized monitor whose state evolves according to the parameter information contained in received events. Our approach, on the contrary, creates many isolated monitor instances, but it maintains indexing information so that it can quickly find relevant monitors. Experiments (Section 5.3) show that our "decentralized-monitoring" strategy performs overall better than the centralized ones. The rest of this section presents two instances of our decentralized monitoring technique, both supported by JavaMOP: one using centralized indexing and the other using decentralized indexing.

Centralized Indexing

Efficient monitor lookup is crucial to reduce the runtime overhead. The major requirement here is to quickly locate all related monitors given a set of parameter instances. Recall that different events can have different sets of parameters: e.g., in Figure 2.1, all three events declare different parameter subsets. Our centralized indexing algorithm constructs multiple indexing trees according to the event definitions to avoid inefficient traversal of the indexes; more specifically, for every distinct set of event parameters found in the specification, an indexing tree is created to map the set of parameters directly into the list of corresponding monitors.

The number and structure of indexing trees needed for a specification can be determined by a simple static analysis of event parameter declarations. For example, for the parametric specification in Figure 2.1, since there are three different sets of event parameters, namely `<v,e>`, `<v>` and `<e>`, three indexing trees will be created to index monitors, as illustrated in Figure 5.4: the first tree uses a pair of `v` and `e` to find the corresponding monitor, while the other two map `v` and, respectively, `e` to the list of related monitors.

We use hash maps in JavaMOP to construct the indexing tree. Figure 5.5 shows the generated monitor look up code for the `updatesource` event in Figure 2.1. This code is inserted at the end of every call to `Vector.add` or other vector changing methods, according to the event definition. One parameter is associated to this event, namely, the vector `v` on which we invoke the method. A map, `UnsafeEnum_v_map`, is created to store the indexing information for `v`, i.e., the $\{v\}$Map in Figure 5.4. When such a method call is encountered during the execution, a concrete vector object will

```
public aspect SafeEnumMonitorAspect {
...
   static Map UnsafeEnum_v_Map = null;
   pointcut UnsafeEnum_updatesource1(Vector v) :
      ((call(* Vector+.add*(..)) || ...)
&& target(v));
   after (Vector v) : UnsafeEnum_updatesource1(v) {
      Object obj = null;
      Map m = UnsafeEnum_v_Map;
      if(m == null) m = UnsafeEnum_v_Map = makeMap(v);
      synchronized(UnsafeEnum_v_Map) {
         obj = m.get(v);
      }
      if (obj != null) {
         synchronized(obj) {
            for(UnsafeEnumMonitor_1 monitor : (List<UnsafeEnumMonitor_1>)obj) {
               monitor.MOP_thisJoinPoint = thisJoinPoint;
               monitor.updatesource(v);
               if(monitor.MOP_match()) {
                  System.out.println(...);
                  monitor.reset();
               }// end of if
            }// end of for
         }// end of synchronized
      }// end of if
   }// end of advice
...
}
```

Figure 5.5: Centralized indexing monitoring code generated by JavaMOP for `updatesource` (from spec in Figure 2.1)

be bound to `v` and the monitoring code will be triggered to fetch the list of related monitors using `UnsafeEnum_v_map`. Then all the monitors in the list will be invoked to process the event.

An important question is when to create a new monitor instance. This is a non-trivial problem in its full generality, because one may need to create "partially instantiated" monitors when events with fewer parameters are observed before events with more parameters. While this partial instantiation can be achieved in a logic-independent manner, motivated by practical needs we adopted a simpler solution in JavaMOP: we let the logic-plugin tell which events are allowed to create new monitors; these events are also required to be parametric by all the specification parameters, such as the `create<v,e>` event in Figure 2.1. All MOP's logic-plugins have been extended to mark their monitor-initialization events. Thus, if an event is generated and a monitor instance for its parameters cannot be found, then a new monitor instance is created for its parameters only if the event is marked; otherwise the event is discarded. This way, no unnecessary monitor instances are created; indeed,

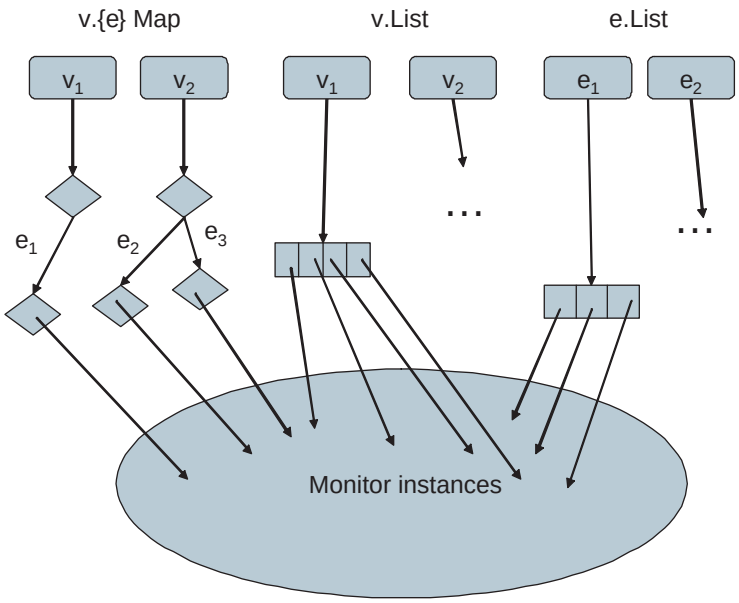

Figure 5.6: Decentralized indexing for monitor in Figure 5.4

it would be pointless and expensive to create monitor instances for all vector updates just because they can be potentially associated with enumerations – monitor instances are created only when enumerations are actually created.

A performance-related concern in our implementation of JavaMOP is to avoid memory leaks caused by hash maps: values of parameters are stored in hash maps as key values; when these values are objects in the system, this might prevent the Java garbage collector from removing them even when the original program has released all references to them. We use weakly referenced hash maps in JavaMOP. The weakly referenced hash map only maintains weak references to key values; hence, when an object that is a key in the hash map dies in the original program, it can be garbage collected and the corresponding key-value pair will also be removed from the hash map. This way, once a monitor instance becomes unreachable, it can also be garbage collected and its allocated memory released.

Note that a monitor instance will be destroyed *only* only when it will never be triggered in the future. Since we have an indexing tree per event parameter set, if a monitor m can potentially be triggered in the future by some event e with a parameter set $(p_1, .., p_n)$, where n can also be 0, then:

1. m appears in the indexing tree corresponding to the parameters $(p_1, .., p_n)$; that is also because of our assumption/limitation that, when m is created, all its possible parameters, including p_1, ..., p_n but potentially more, were available; when m was created, it was added to all the

indexing trees corresponding to (subsets of) its parameters, including that of $(p_1, .., p_n)$; and

2. if e is ever generated in the future, m will be referred from the indexing tree for $(p_1, .., p_n)$. This is because if e really occurs at some moment in the future, then p_1, .., p_n should all be live objects and thus the mapping in the corresponding indexing tree has not been destroyed.

Therefore, if a future event can ever trigger m, then m is not garbage collectible. This guarantees the soundness of our usage of weak references. One interesting corner case here is when n is 0, i.e., when some event has no parameter. In such case, the corresponding indexing tree (for the empty set of parameters) is actually a list instead of a map. Thus, even if all parameters die, the monitor will still be kept alive because there is a reference to it in that list. But this only happens when at least one of the events in the specification has no parameters.

Optimization: Decentralized Indexing

The centralized-indexing-decentralized-monitor approach above can be regarded as a centralized database of monitors. This solution proves to be acceptable wrt runtime overhead in many of the experiments that we carried out; in particular, it compares favorably with centralized-monitor approaches (see Section 5.3). However, reducing runtime overhead is and will always be a concern in runtime verification. We next propose a further optimization based on decentralizing indexing. This optimization is also implemented in JavaMOP.

In *decentralized indexing*, the indexing trees are piggybacked into states of objects to reduce the lookup overhead. For every distinct subset of parameters that appear as a parameter of some event, JavaMOP automatically chooses one of the parameters as the *master parameter* and uses the other parameters, if any, to build the indexing tree using hash maps as before; the resulting map will then be declared as a new field of the master parameter. For example, for the `updatesource` event in Figure 2.1, since it has only the `<v>` parameter, `v` is selected as master parameter and a new field will be added to its `Vector` class to accommodate the list of related monitor instances at runtime. Figure 5.6 shows the decentralized version of the centralized indexing example in Figure 5.4, and Figure 5.7 shows the generated decentralized indexing monitoring code for the `updatesource` event.

Comparing Figures 5.7 and 5.5, one can see that the major difference between the centralized and the decentralized indexing approaches is that the list of monitors related to `v` can be directly retrieved from `v` when using decentralized indexing; otherwise, we need to look up the list from a hash map. Decentralized indexing thus scatters the indexing over objects in the system and avoids unnecessary lookup operations, reducing both runtime overhead and memory usage. It is worth

```
public aspect SafeEnumMonitorAspect {
...
   List Vector.UnsafeEnum_v_List = null;
   pointcut UnsafeEnum_updatesource1(Vector v) :
      ((call(* Vector+.add*(..))
|| ...) && target(v));
   after (Vector v) : UnsafeEnum_updatesource1(v) {
      Object obj = null;
      if(v.UnsafeEnum_v_List == null) v.UnsafeEnum_v_List = makeList();
      {
          obj = v.UnsafeEnum_v_List;
      }
      if (obj != null) {
         synchronized(obj) {
            for(UnsafeEnumMonitor_1 monitor : (List<UnsafeEnumMonitor_1>)obj) {
               monitor.MOP_thisJoinPoint = thisJoinPoint;
               monitor.updatesource(v);
               if(monitor.MOP_match()) {
                  System.out.println(...);
                  monitor.reset();
               }// end of if
            }// end of for
         }// end of synchronized
      }// end of if
   }// end of advice
...
}
```

Figure 5.7: Decentralized indexing monitoring code automatically generated by JavaMOP for `updatesource`

noting that decentralized indexing does *not* affect the behavior of disposing unnecessary monitor instances as discussed in the previous section: when an object is disposed, all the references to monitor instances based on this object will also be discarded, no matter whether they are stored in maps using weak references or whether they are embedded as fields of the object.

On the negative side, decentralized indexing involves more instrumentation than the centralized approach, sometimes beyond the boundaries of the monitored program, since it needs to modify the original signature of the master parameter: for the monitoring code in Figure 5.7, the Java library class `Vector` has to be instrumented (add a new field). This is usually acceptable for testing/debugging purposes, but may not be appropriate if we use MOP as a development paradigm and thus want to leave monitors as part of the released program. If that is the case, then one should use centralized indexing instead, using the attribute `centralized`.

The choice of the master parameter may significantly affect the runtime overhead. In the specification in Figure 2.1, since there is a one-to-many relationship between vectors and enumerations,

it would be more effective to choose the enumeration as the master parameter of the `create` event. Presently, JavaMOP picks the first parameter encountered in the analysis of the MOP specification as the master parameter for each set of event parameters. Hence, the user can control the choice of the master parameter by putting, for each set of parameters P, the desired master parameter first in the list of parameters of the first event parametric over P.

5.3 Evaluation

We have applied JavaMOP on tens of programs, including several large-scale open source programs, e.g., the DaCapo benchmark suite [18], the Tracematches benchmark suite [12], and Eclipse [36]. Our evaluation mainly focuses on two aspects: the expressivity of the specification language and the runtime overhead of monitoring. The properties used in our experiments come from two sources: properties used in other works (e.g., [44, 66, 12, 22]) and our own formalization of informal descriptions in software documentation.

With the currently supported logic-plugins and the generic support for parameters, JavaMOP is able to formally and concisely express most of the collected properties. One interesting exception is the SQL injection from PQL [66], which we implemented using the raw MOP specification shown in Figure 5.3. A large portion, nearly half, of the properties that we have tried are recoverable/enforceable. Many violations of properties were revealed in our experiments, although we did not focus on error detection; when violations occurred, we were able to quickly locate their causes using JavaMOP. The rest of this section focuses on performance evaluation, on discussing some of the detected violations, and on current limitations of our implementation.

5.3.1 Performance Evaluation

The monitoring code generated by JavaMOP caused low runtime overhead, below 10%, in most experiments even with centralized indexing. By turning on the decentralized indexing, few experiments showed noticeable runtime overhead. In what follows, we evaluate JavaMOP's runtime overhead using the DaCapo benchmark, and also compare JavaMOP with other runtime verification techniques, namely, Tracematches and PQL.

Our experiments were carried out on a machine with 1.5GB RAM and Pentium 4 2.66GHz processor. The operating system used was Ubuntu Linux 7.10. We used the DaCapo benchmark version 2006-10; it contains eleven open source programs [18]: antlr, bloat, chart, eclipse, fop,

hsqldb, jython, luindex, lusearch, pmd, and xalan. The provided default input was used together with the -converge option to execute the benchmark multiple times until the execution time falls within a coefficient of variation of 3%. The average execution time of six iterations after convergence are then used to compute the runtime overhead. Therefore, the runtime overhead percentages in Tables 5.3 and 5.4 and should be read "±3" (meaning negative numbers are possible).

Properties

The following general properties, some of which were borrowed from [22], were checked using JavaMOP:

1. UnsafeEnum: Do not update `Vector` while enumerating its elements using the `Enumeration` interface (Figure 2.1).
2. UnsafeMapIter: Do not update a `Map` when using the `Iterator` interface to iterate its contents (Figure 5.2).
3. HashMap: The hash code of an object should not be changed when the object is used as a key in a hash map.
4. HasNext: Always call the `hasNext()` method of an iterator before calling its `next()` method.
5. SafeFileWriter: SafeFileWriter ensures that all writes to a `FileWriter` happen between creation and close of the `FileWriter`, and that the creation and close events are matched pairs.
6. LeakingSync: Only access a `Collection` via its synchronized wrapper once the wrapper is generated by the `Collections.synchronized*` methods. Note that the original LeakingSync specified in [22] *only* allows synchronized accesses to synchronized collections. This causes spurious failures because the synchronized methods call the unsynchronized versions. Our version improves it by allowing calls to the unsynchronized methods so long as they happen within synchronized calls. It cannot be expressed in Tracematches because it is not a regular pattern.

More properties have been checked in our experiments; we choose these six properties to include in this thesis because they generate a comparatively larger runtime overhead. We excluded those with little overhead. Three of these properties are recoverable: `HashMap` (the monitor can maintain a shadow map based on `IdentityHashMap` as backup), `HasNext` (make a call to `hasNext()`

before `next()`), and `LeakingSync` (redirect call to the synchronized wrapper). `SafeFileWriter` and `LeakingSync` are CFG properties and others are ERE properties.

For every property, we provided three MOP specifications: a formal specification for decentralized indexing, the same formal specification for centralized indexing, and a (hand-optimized) raw MOP specification. The last one is supposedly the best monitoring code for that property and was used to evaluate the effectiveness of our monitor generation algorithm. The AspectJ compiler 1.5.3 (AJC) was used in these experiments to compile the generated monitoring AspectJ code.

Statistics and Results of the Evaluation

	UnsafeEnum	UnsafeMapIter	HashMap	HasNext	SafeFileWriter	LeakingSync
DaCapo	1147	6663	1729	2639	2966	12855

Table 5.1: Instrumentation statistics: instrumentation points in the DaCapo benchmark

	UnsafeEnum		UnsafeMapIter		HashMap		HasNext		SafeFileWriter		LeakingSync	
antlr	10K	0	1K	0	0	0	0	0	0	0	8472	0
bloat	0	0	90M	1M	391K	46K	155M	1M	385	231	5.5M	0
chart	57	0	569K	815	8K	3K	6K	815	0	0	634K	0
eclipse	16K	0	38K	31	31K	19K	1K	31	0	0	74K	0
fop	7	1	49K	79	17K	6K	277	79	0	0	182K	0
hsqldb	174	0	0	0	0	0	0	0	0	0	0	0
jython	50K	0	174K	50	443	439	106	50	0	0	23M	0
luindex	457K	14K	82K	8K	9K	9K	28K	8K	0	0	1.5M	0
lusearch	335K	0	405K	0	416	416	0	0	0	0	1.2M	0
pmd	717	0	25M	1M	11K	105	46M	8M	32	20	26M	0
xalan	5K	0	199K	0	124K	78K	0	0	0	0	5M	0

Table 5.2: Monitoring statistics: generated events(left column) and monitor instances(right column). K = $\times 10^3$, M = $\times 10^6$

Tables 5.1 and 5.2 show the instrumentation and monitoring statistics for monitoring the above properties in DaCapo: Table 5.1 gives the number of points statically instrumented for monitoring each of the properties; Table 5.2 gives the number of events and the number of monitor instances generated at runtime using centralized indexing. Both these numbers are collected from a single execution of the benchmark. The first row in each table gives the names of the properties, and the first column in Table 5.2 gives the programs. We do not split the static instrumentation points by different programs because they are merged together in the benchmark suite; some of them even share common packages. Decentralized indexing does not change the number of generated events or monitor instances; it only affects the monitor indexing.

	UnsafeEnum			UnsafeMapIter			HashMap		
antlr	0.0	0.0	1.5	0.0	0.0	0.0	0.0	0.0	1.1
bloat	2.4	0.0	0.0	**385**	**176**	**24.2**	2.4	1.8	1.4
chart	0.0	0.0	0.0	0.3	0.0	0.0	4.8	3.6	4.8
eclipse	2.4	4.1	0.8	0.0	0.0	1.4	3.6	3.7	0.5
fop	0.4	1.2	0.6	1.7	1.5	0.0	0.0	0.0	0.0
hsqldb	0.0	3.3	0.0	0.0	0.9	1.2	0.0	0.0	2.1
jython	0.5	0.6	0.0	1.6	0.8	0.5	0.7	0.2	0.3
luindex	2.6	1.6	0.2	3.2	1.9	0.5	0.6	1.2	1.8
lusearch	6.6	0.5	0.0	9.5	0.0	0.0	0.0	0.0	0.0
pmd	0.0	0.0	0.0	**272**	**44.8**	**11.3**	0.5	0.0	0.0
xalan	0.0	3.5	4.4	4.8	6.7	5.4	7.2	4.7	6.5

	HasNext			SafeFileWriter			LeakingSync		
antl	0.0	0.4	0.0	0.9	1	0.5	2.7	0.0	0.0
bloat	**323**	**154**	**36.3**	0.1	27	3.2	13.5	3.2	2.2
chart	0.0	0.0	0.0	0.1	0.1	0.2	0.1	0.5	0.0
eclipse	0.0	3.8	1.5	0.2	-2	0.2	0.8	3.0	3.1
fop	1.7	0.8	1.5	-1.0	-2.0	1.0	14.7	0.5	1.0
hsqldb	0.0	0.8	0.0	0.0	2	1.0	1.1	1.4	1.4
jython	1.3	0.0	0.6	0.2	1	-1.0	30.2	0.0	2.3
luindex	0.9	0.3	0.0	0.0	0.0	0.0	4.3	3.2	2.2
lusearch	0.3	0.0	**32.4**	0.2	0.1	0.3	32.4	1.1	0.6
pmd	**353**	**25.4**	**13.7**	1.0	-2	0.3	34.3	5.4	8.0
xalan	0.0	2.8	3.0	0.1	-2.5	0.0	3.0	1.5	1.7

Table 5.3: Runtime overhead (in percentage; e.g., 14.7 means 14.7% slower) of JavaMOP: centralized | decentralized | raw

These two tables show that the properties selected in our experiments imposed heavy runtime monitoring on the programs: a large number of points, ranging from one thousand to twelve thousand, in the original programs were instrumented to insert the monitoring code. The monitoring code was frequently triggered during the execution, especially for those properties involving the Java `Collection` classes, e.g., `UnsafeMapIter`, `HashMap`, and `HasNext`. Some properties generated numerous runtime checks but only a few, even zero, monitor instances were created (e.g., `UnsafeEnum` and `LeakingSync`). The reason is that these properties observe some frequently visited methods, but the events that we allowed to create monitor instances rarely or never occurred. For example, `LeakingSync` checks all the method calls on the `Collection` interface, but no calls to `Collections.synchronized*` methods happened in these experiments, so no monitor-initialization events were created. Such experiments are particularly useful to evaluate the effectiveness of the generated monitoring code to filter dynamically irrelevant events, i.e., events that have no effect on the current monitor states. Also, a big difference between the number of events and the number of created monitor instances (e.g., jython-`UnsafeEnum` and bloat-`LeakingSync`) indicates a real

potential for static analysis optimizations.

Table 5.3 summarizes the runtime overhead measured in our experiments, represented as a *slow-down percentage* of the monitored program over the original program. For every property-program combination, three monitoring runtime overhead numbers are given: with centralized indexing, with decentralized indexing, and using a hand-optimized raw MOP specification. Among all 66 experiments (recall that we already excluded some results with little overhead), only 11 (bold) caused more than 10% slow-down with centralized indexing; for the decentralized indexing version, this number reduces to 4. Except for the 4 worst cases, with decentralized indexing JavaMOP generates monitoring code *almost as efficient as the hand-optimized code.*

Analyzing Tables 5.3 and 5.2, one can see that decentralized indexing handles the dynamically irrelevant events much better than centralized indexing, e.g., when checking the `LeakingSync` property. This is caused by the fact that, when there is no related monitor instance, decentralized indexing only checks an object field, while centralized indexing needs to make an expensive hash map lookup. The runtime overhead is determined not only by the frequency of reaching monitoring code, but also by the execution time of the monitored action. For example, `HashMap` required quite heavy monitoring on many programs but did not cause any noticeable performance impact. This is because the methods checked for `HashMap`, including `put`, `remove`, and `contains`, are relatively slow. On the other hand, checking `bloat` and `pmd` against `UnsafeMapIter` and `HasNext` is as bad as it can be: the monitored actions take very little time to execute (e.g., the `hasNext` and `next` methods of `Iterator`) and they are used very intensively during the execution (indicated by the massive numbers in Table 5.2). Even for such extreme cases, the monitoring code generated by JavaMOP with decentralized indexing may be considered acceptable: slowdown between 2 and 3 times. However, one can always choose to implement a hand-optimized raw MOP specification for the property of interest; in our case, the raw MOP specification reduced the runtime overhead to only 20-30%.

Comparing JavaMOP, Tracematches, and PQL

Attempts have also been made to compare JavaMOP with other existing trace monitoring tools. However, some of them are not publicly available, others have limitations that prevented us from using them in our experiments. Consequently, we only succeeded to compare JavaMOP thoroughly with Tracematches and partially with PQL.

Table 5.4 compares the percent overheads of JavaMOP, PQL, and Tracematches. N/E refers

to specifications that were not expressible. Negative numbers can be attributed to the 3% noise in the measurements. Tracematches is unable to support LeakingSync because the property is truly context-free. PQL is also unable to support it because it requires events corresponding to the beginning and end of synchronized method calls, and PQL can only trigger events on the end of method calls.

	UnsafeEnum			UnsafeMapIter			HashMap		
	MOP	PQL	TM	MOP	PQL	TM	MOP	PQL	TM
antlr	2	82	0	2	82	-2	3	6	0
bloat	627	8694	11258	925	crashes	>10000	14	9	-2
chart	2	50	11	0	50	-1	-1	1	-1
eclipse	-2	1	2	1	1	8	0	1	1
fop	-1	24	5	-3	24	11	3	2	0
hsqldb	0	78	17	0	78	29	0	3	15
jython	0	12	16	7	12	57	0	23	15
luindex	3	181	9	5	181	7	1	8	1
lusearch	4	132	34	-1	132	9	1	1	8
pmd	178	1334	175	196	crashes	>10000	-1	0	3
xalan	1	53	10	4	53	10	0	5	1

	HasNext			SafeFileWriter			LeakingSync		
	MOP	PQL	TM	MOP	PQL	TM	MOP	PQL	TM
antlr	1	2	3	2	22	N/E	1	N/E	N/E
bloat	1112	5929	2452	27	97	N/E	13	N/E	N/E
chart	-1	3	0	0	37	N/E	4	N/E	N/E
eclipse	0	2	-1	-2	1	N/E	1	N/E	N/E
fop	0	2	-1	-2	47	N/E	1	N/E	N/E
hsqldb	0	6	15	2	95	N/E	1	N/E	N/E
jython	0	0	13	1	crashes	N/E	41	N/E	N/E
luindex	-2	93	2	0	33	N/E	1	N/E	N/E
lusearch	-1	59	9	0	49	N/E	2	N/E	N/E
pmd	191	1870	52	-2	658	N/E	36	N/E	N/E
xalan	0	0	2	-2.5	164	N/E	3	N/E	N/E

Table 5.4: Average percent runtime overhead for JavaMOP CFG (MOP), PQL, and Tracematches (TM) (convergence within 3%); N/E means "not expressible".

Table 5.4 shows that JavaMOP generates more efficient monitoring code than Tracematches and PQL, often close to the hand-optimized code when using decentralized indexing. Since JavaMOP generates *standard* AspectJ code, it gives us the freedom to choose off-the-shelf compilers. In our experiments, ABC tended to take more time to compile the code than AJC, e.g., it took ABC nearly an hour to compile Aprove but AJC needed only a few minutes. Also, JavaMOP provides better expressivity than both Tracematches and PQL thanks to JavaMOP's genericity in logics.

PQL and Tracematches have their own strengths and the above comparison should not be in-

terpreted as an argument against them. PQL provides a general specification formalism extending context-free grammars; it is therefore not surprising that it generates a larger runtime overhead. Tracematches implements a sound and specialized algorithm to support universally quantified regular patterns. The parametric framework discussed in our paper is generic and logic-independent, therefore the present JavaMOP implementation does not provide any logic-specific optimizations or specializations like those in Tracematches.

5.3.2 Violation Detection

As mentioned, error detection was not the main focus in our experiments; we consider that, for error detection, runtime verification needs to be combined with test case generation. However, we still encountered unexpectedly many violations during the evaluation of JavaMOP. One reason is that many safety properties in our experiments were devised for checking performance, and are therefore not strictly required to hold in all programs. Consequently, many violations do not lead to actual errors in the program. For example, violations of the `hasNext` property were found in some Java library classes, e.g., `AbstractCollection` and `TreeMap`. It turned out that these implementations use the size of the collection instead of the `hasNext` method to guard the iteration of elements. We also found violations indicating possible semantic problems of programs, which are subtle and thus difficult to find by ordinary testing. We next discuss some of these.

Potential Errors.

There is a known problem in jHotDraw about using objects of `Enumeration`: one can edit a drawing, which may update a vector in the program, while making the animation for the drawing, which uses an enumerator of the vector. As expected, JavaMOP was able to find this problem.

We also found violations of some *interface contracts*, i.e., rules to use interfaces, in Eclipse. These can lead to resource leaks as pointed out in [44] and [66]. Three kinds of properties were checked in our Eclipse experiments:

1. The `dispose` method needs to be called to release acquired resources before a GUI widget is finalized.

2. The `remove*Listener` should be called by a host object to notify its listeners (registered by calling `add*Listener`) to release resources before it is finalized. `*` represents the name of the listener.

3. Eclipse uses Lucene [65] as its search engine; in Lucene, it is required that, before a `Dir` object is closed (by calling its `close` method), all the file readers created by the `Dir` object should be closed.

We instrumented the GUI package of Eclipse with these three properties and also the JDT package with the second property (note that there are many different `add*Listener-remove*Listener` pairs in these two packages). Then we used the instrumented Eclipse in our development work (no noticeable slow-down was experienced during the evaluation). More than 30 violations were detected in the GUI package, while none was found in the JDT package – this may indicate the importance of the second property. In summary, the GUI package, which is more complex and harder to test, seems less reliable w.r.t. to memory leaks.

Inappropriate Programming Practice

Several unexpected violations were encountered during our experiments. For example, we ran into some violations in Xalan [90] when checking a simple property about the `Writer` class in Java: no writes can occur after the writer is closed (by calling the `close` method). This is, according to the Java documentation which states that an exception should be raised, a must-have property. Despite these violations, no errors occurred in Xalan. Using JavaMOP, we located the places causing the violations without much insight of the program and a quick review showed that a pool of writer instances is used in Xalan to avoid unnecessary re-creations, but the writer can be closed before it is returned to the pool. However, the program uses `StringWriter`, whose `close` method happens to have no effect. Although it is not an error in this implementation, we believe that it is inappropriate programming practice: the writer should be cleared instead of closed when returned to the pool.

5.3.3 Limitations of MOP and JavaMOP

The current MOP logic-plugins encapsulate monitor synthesis algorithms only for non-parametric trace logics. Even though the new MOP specification language allows universal parameters to be added to any of these logics, there is no way to add nested parameters, or existential ones. We intend to soon add a logic-plugin for Eagle [15], a "super-logic" generalizing both ERE and LTL, and also allowing arbitrary quantification and negation, but do not expect it to have a stimulating runtime overhead.

The gap between dynamic events for monitoring and static monitor integration based on AOP can lead to some limitations of MOP tools. Ideally, for variable update events, the MOP tool should

instrument all the updates of involved variables. But, statically locating all such updates requires precise alias analysis. Therefore, JavaMOP only allows update events for variables of primitive types. In addition, static instrumentation may cause extra performance penalty of monitoring. For the specification in Figure 2.1, one can see that the monitor is not "interested" in `next` events after `create` until an `updatesource` event is encountered. But since we instrument the program statically, the monitor keeps receiving `next` events even when they are not needed. These limitations may be relaxed by utilizing dynamic AOP tools, but more discussion on this direction is out of the scope of this paper. However, since MOP can also be used to add new functionality to a program, one may not want to miss any related event: some action may be executed even when the event does not affect the monitor state.

Chapter 6

Predictive Runtime Analysis

In this section, we present the predictive runtime analysis based on sliced causality. We first recall the previous work on happen-before causality and introduce a paramtric framework for defining and proving feasible causalities, which naturally captures the existing happend-before causality. We then define sliced causality and prove its soundness within the parametric causality framework . We also propose an algorithm to efficiently compute sliced causality from multithreaded program executions using vector clocks. This algorithm is combined with lock-atomicity to achieve even more powerful predictive runtime analysis.

6.1 Happen-Before Causalities

The first happen-before relation was introduced almost 3 decades ago by Lamport [63], to formally model and reason about concurrent behaviors of distributed systems. Since then, a plethora of variants of happen-before causal partial order relations have been introduced in various frameworks and for various purposes. The basic idea underlying happen-before relations is to observe the events generated by the execution of a distributed system and, based on their order, their type and a straightforward causal flow of information in the system (e.g., the receive event of a message follows its corresponding send event), to define a partial order relation, the happen-before causality. Two events related by the happen-before relation are causally linked in that order.

When using a particular happen-before relation for (concurrent) program analysis, the crucial property of the happen-before relation is that, for an observed execution trace τ, other *sound permutations* of τ, also called *linearizations* or *linear extensions* or *consistent runs* or even *topological sortings* in the literature, are also possible computations of the concurrent system. Consequently, if any of these linearizations violates or satisfies a property φ, then the system can indeed violate or satisfy the property, regardless of whether the particular observed execution that generated the happen-before relation violated or satisfied the property, respectively. For example, [33] defines

formulae *Definitely*(φ) and *Possibly*(φ), which hold iff φ holds in all and, respectively, in some possible linearizations of the happen-before causality.

The soundness/correctness of a happen-before causality can be stated as follows: given a happen-before causal partial order extracted from a run of the concurrent system under consideration, all its linearizations are *feasible*, that is, they correspond to other possible execution of the concurrent system. To prove it, one needs to formally define the actual computational model and what a concurrent computation is; these definitions tend to be rather intricate and domain-specific. For that reason, proofs need to be redone in different settings facing different "details", even though they follow conceptually the same idea. In the next section we present a simple and intuitive property on traces, called *feasibility*, which ensures the desired property of the happen-before causality and which appears to be easy to check in concrete situations.

To show how the various happen-before causalities fall as special cases of our parametric approach, we recall two important happen-before partial orders, one in the context of distributed systems where communication takes place exclusively via message passing, and another in the context of multithreaded systems, where communication takes place via shared memory. In the next section we show that their correctness [24, 82] follow as corollaries of our main theorem. In Section 6.3 we define another happen-before causality, called *sliced causality*, which non-trivially uses static analysis information about the multithreaded program. The correctness of sliced causality will also follow as a corollary of our main theorem in the next section.

In the original setting of [63], a distributed system is formalized as a collection of processes communicating only by means of asynchronous message passing. A process is a sequence of events. An event can be a *send* of a message to another process, a *receive* of a message from another process, or an *internal* (local) event.

Definition 28 *Let τ be an execution trace of a distributed system consisting of a sequence of events as above. Let E be the set of all events appearing in τ and let the* ***happen-before*** *partial order "$\rightarrow$" on E be defined as follows:*

1. *if e_1 appears before e_2 in some process, then $e_1 \rightarrow e_2$;*
2. *if e_1 is the send and e_2 is the receive of the same message, then $e_1 \rightarrow e_2$;*
3. *$e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ implies $e_1 \rightarrow e_3$.*

A space-time diagram to illustrate the above definition is shown in Figure 6.1 (A), in which e_1 is a send message and e_2 is the corresponding receive message; $e_1 \rightarrow e_2$ and $e_1 \rightarrow e_3$, but e_2

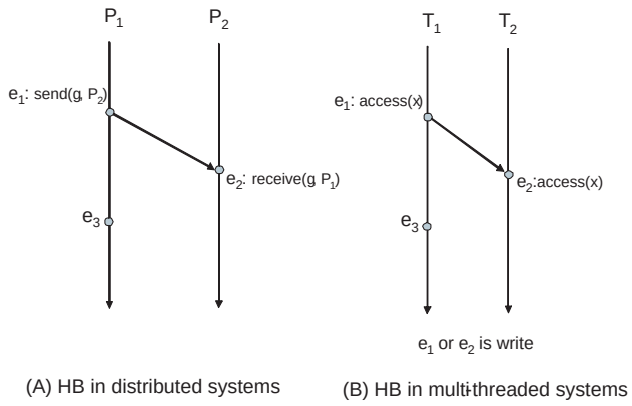

Figure 6.1: Happen-before partial-order relations

and e_3 are not related. It is easy to prove that $(E, \rightarrow)$ is a partial order. The soundness of this happen-before relation, i.e., all the permutations of τ consistent with $\rightarrow$ are possible computations of the distributed system, was proved in [24] using a specific formalization of the global state of a distributed system. This property will follow as an immediate corollary of our main theorem in the next section.

Happen-before causalities have been devised in the context of multithreaded systems for various purposes. For example, [71, 72] propose datarace detection techniques based on intuitive multi-threaded variants of happen-before causality, [82] proposes a happen-before relation that drops read/read dependencies, and [83] even drops the write/write conflicts but relates each write with all its subsequent reads atomically.

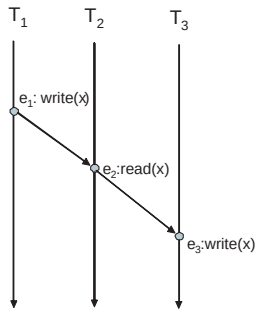

Figure 6.2: Happen-before causality in multi-threaded systems

Finding appropriate happen-before causalities for multithreaded systems is a non-trivial task. The obvious approach would be to map the inter-thread communication in a multi-threaded system into send/receive events in some corresponding distributed system. For example, starting a new

thread generates two events, namely a send event from the parent thread and a corresponding receive event from the new thread; releasing a lock is a send event and acquiring a lock is a receive event. A write on a shared variable is a send event while a read is a receive event. However, such a simplistic mapping suffers from several problems related to the semantics of shared variable accesses. First, every write on a shared variable can be followed by multiple reads whose order should not matter; in other words, some "send" events now can have multiple corresponding "receive" events. Second, consider the example in Figure 6.2. Since e_3 may write a different value into x other than e_1, the value read at where e_2 occurs may change if we observe e_3 after e_1 but before e_2 appears. Therefore, $e_1e_3e_2$ may not be a trace of some feasible execution since e_2 will not occur any more. Hence, a causal order between e_2 and e_3 should be enforced, which cannot be captured by the original definition in [63].

The various causalities for multithreaded systems address these problems (among others). However, each of them still needs to be proved correct: any sound permutation of events results in a feasible execution of the multithreaded system. If one does not prove such a property for one's desired happen-before causality, then one's analysis techniques can lead to false alarms. We next recall one of the simplest happen-before relations for multi-threaded systems [82]:

Definition 29 *Let τ be an execution of a multithreaded system, let E be the set of all events in τ, and let the* ***happen-before*** *partial order "$\rightsquigarrow$" on E be defined as follows:*

1. *if e_1 appears before e_2 in some thread, then $e_1 \rightsquigarrow e_2$;*
2. *if e_1 and e_2 are two accesses on the same shared variable such that e_1 appears before e_2 in the execution and at least one of them is a write, then $e_1 \rightsquigarrow e_2$;*
3. *$e_1 \rightsquigarrow e_2$ and $e_2 \rightsquigarrow e_3$ implies that $e_1 \rightsquigarrow e_3$.*

In Figure 6.1 (B), $e_1 \rightsquigarrow e_2$ and $e_1 \rightsquigarrow e_3$, but e_2 and e_3 are not comparable under $\rightsquigarrow$.

6.2 Parametric Framework for Causality

We here define a parametric framework that axiomatizes the notion of causality over events and *feasibility* of traces in a system-independent manner. We show that proving the feasibility of the linearizations of a causal partial order extracted from an execution can be reduced to checking a simpler "closure" local property on feasible traces.

Let *Events* be the set of all events. A trace τ is a finite ordered set of (distinct) events $\{e_1 < e_2 < \cdots < e_n\}$, usually identified with the *Events** word $e_1 \cdots e_n$. Let $\xi_\tau = \{e_1, e_2, \ldots, e_n\}$ be called the alphabet of τ and $<_\tau$ be the total order on ξ_τ induced by τ. Let *Traces* denote the set of all such traces. Given a set X, let $\mathcal{PO}(X)$ denote the set of partial orders defined on subsets of X, that is, the set of pairs $(\xi, <)$ where $\xi \subseteq X$ and $< \subseteq \xi \times \xi$ is a partial order.

Definition 30 *A* causality operator *is a partial function* C : *Traces* $\nrightarrow \mathcal{PO}$(*Events*) *s.t.:*

1. *If* $C(\tau) = (\xi, <)$, *then* $\xi = \xi_\tau$ *and* $< \subseteq <_\tau$; *and*
2. *For any* $\tau = \tau_1 e_1 e_2 \tau_2$ *such that* $C(\tau) = (\xi, <)$ *with* $e_1 \not< e_2$, $C(\tau_1 e_2 e_1 \tau_2)$ *is also defined and equal to* $C(\tau)$.

Let Dom(C) *denote the domain of* C.

Lemma 3 *If* $C(\tau) = (\xi, <)$, *then for any linearization* τ' *of* $C(\tau)$, $C(\tau')$ *is defined and equal to* $C(\tau)$.

Proof: Any permutation of a trace is a product of adjacent transpositions [56]. So τ' can be achieved by a sequence of transpositions from τ, each of which exchanges two adjacent events $e_1, e_2 \in \xi$ such that $e_1 \not< e_2$. Therefore, it is straightforward to prove the lemma by induction on the number of transpositions using condition (ii) in Defintion 30. □

Note that condition (ii) in the definition above is closely related to that of trace equivalence introduced by Mazurkiewicz in [50]. However, the theory of Mazurkiewicz traces starts with a given dependency relation on events and considers equivalence of traces according to that fixed dependency, while in our framework, we prefer to associate a separate dependency relation to each trace, assuming that only at runtime we can get enough information about the causality for a given trace; for example, acquiring lock l and writing shared variable x can be or not dependent events, depending on the particular execution of the program. This allows us to be more precise while still using part of the generic Mazurkiewicz trace theory to simplify our correctness proofs.

Any concurrent system can produce only a particular subset of *feasible* traces, which are in the following relationship with the corresponding causality operator:

Definition 31 *Given a causality operator* C, *a set* $\mathcal{F}$ *of traces is* C-feasible *iff* C *is defined on* $\mathcal{F}$ *and for any* $\tau \in \mathcal{F}$ *with* $C(\tau) = (\xi, <)$, $\mathcal{F}$ *contains all the linearizations of* $C(\tau)$ *(i.e., all traces* τ' *such that* $\xi_{\tau'} = \xi$ *and* $< \subseteq <_{\tau'}$*).*

Theorem 5 *$\mathcal{F}$ is C-feasible iff C is defined on $\mathcal{F}$ and for any $\tau = \tau_1 e_1 e_2 \tau_2 \in \mathcal{F}$ such that $C(\tau) = (\xi, <)$, $e_1 \not< e_2$ implies $\tau_1 e_2 e_1 \tau_2 \in \mathcal{F}$.*

Proof: if $\mathcal{F}$ is C-feasible iff C is defined on $\mathcal{F}$ then it is obvious that for any $\tau = \tau_1 e_1 e_2 \tau_2 \in \mathcal{F}$ such that $C(\tau) = (\xi, <)$, $e_1 \not< e_2$ implies $\tau_1 e_2 e_1 \tau_2 \in \mathcal{F}$ according to Definition 31. If for any $\tau = \tau_1 e_1 e_2 \tau_2 \in \mathcal{F}$ such that $C(\tau) = (\xi, <)$, $e_1 \not< e_2$ implies $\tau_1 e_2 e_1 \tau_2 \in \mathcal{F}$, then for any linearization τ of $C(\tau)$, $\tau' \in \mathcal{F}$, using the same observation in Lemma 3. Hence, $\mathcal{F}$ is C-feasible. □

Corollary 7 *$Dom(C)$ is C-feasible, More precisely, if $C(\tau) = (\xi, <)$ then for any τ', $C(\tau') = C(\tau)$ if and only if $\xi_{\tau'} = \xi$ and $< \subseteq <_{\tau'}$.*

Proof: The result follows immediately from Definition 30 and Theorem 5. □

The two variants of happen-before relations discussed in Section 6.1 can be captured as instances of our parametric framework. For the happen-before relation defined in Definition 28, let Events_{hb} be the set of all the send, receive and internal events.

Corollary 8 *For an observed trace τ, any permutation of τ consistent with $\rightarrow$ is a possible computation of the distributed system.*

Proof: Let $\mathcal{C}_{hb}$ be the partial function $\mathit{Traces}_{hb} \twoheadrightarrow \mathcal{PO}(\mathit{Events}_{hb})$ with $\mathcal{C}_{hb}(\tau) = (\xi_\tau, \rightarrow)$ for any $\tau \in \mathit{Traces}_{hb}$. Let $\mathcal{F}_{hb}$ be the set of computation traces of the distributed system as defined in [24]. The result follows from Theorem 5, noticing that $\mathcal{C}_{hb}$ is a causality operator and $\mathcal{F}_{hb}$ is $\mathcal{C}_{hb}$-feasible. □

For the happen-before relation in Definition 29, let Events_{mhb} be the set of all the write and read events on shared variables as well as all internal events.

Corollary 9 *For an observed trace τ of a multi-threaded system, any permutation of τ consistent with $\rightsquigarrow$ is a possible execution of the multi-threaded system.*

Proof: Let $\mathcal{C}_{mhb}$ be the partial function $\mathit{Traces}_{mhb} \twoheadrightarrow \mathcal{PO}(\mathit{Events}_{mhb})$ with $\mathcal{C}_{mhb}(\tau) = (E_\tau, \rightsquigarrow)$ for any $\tau \in \mathit{Traces}_{mhb}$. Let $\mathcal{F}_{mhb}$ be the set of traces that are generated by all feasible executions of the multi-threaded system (see, e.g., [82] for a formalization of multi-threaded systems). The result follows from Theorem 5, noticing that $\mathcal{C}_{mhb}$ is a causality operator and $\mathcal{F}_{mhb}$ is $\mathcal{C}_{mhb}$-feasible. □

6.3 Sliced Causality

Without additional information about the structure of the program that generated the event trace τ, the least restrictive causal partial order that an observer can extract from τ is the one which is total on the events generated by each thread and in which each write event of a shared variable precedes all the corresponding subsequent read events. This is investigated and discussed in detail in [83]. In what follows we show that one can construct a much more general causal partial order, called *sliced causality*, by making use of dependence information obtained statically and dynamically. Briefly, instead of computing the causal partial order on all the observed events like in the traditional happen-before based approaches, our approach first slices τ according to the desired property and then computes the causal partial order on the achieved slice; the slice contains all the events relevant to the property, as well as all the events upon which the relevant events depend. This way, irrelevant causality on events is trimmed without breaking the soundness of the approach, allowing more permutations of relevant events to be analyzed and resulting in better coverage of the analysis.

We employ dependencies among events to assure the correct slicing. The dependence discussed here somehow relates to *program slicing* [51], but we focus on finer grained units here, namely events, instead of statements. Our analysis keeps track of actual memory locations in every event, available at runtime, avoiding inter-procedural analysis. Also, we need *not* maintain the entire dependence relation, since we only need to compute the causal partial order among events that are relevant to the property to check. This leads to an effective vector clock (VC) algorithm ([28]).

Intuitively, event e' *depends upon* event e in τ, written $e \sqsubset e'$, iff a change of e may change or eliminate e'. This tells the observer that e *should occur before* e' *in any consistent permutation of* τ. There are two kinds of dependence: (1) *control dependence*, written $e \sqsubset_{ctrl} e'$, when a change of the state of e may eliminate e'; and (2) *data-flow dependence*, written $e \sqsubset_{data} e'$, when a change of the state of e may lead to a change in the state of e' . While the control dependence only relates events generated by the same thread, the data-flow dependence may relate events generated by different threads: e may write some shared variable in a thread t, which is then read in another thread t'.

6.3.1 Events and Traces

Events represent atomic steps observed in the execution of the program. In this paper, we focus on multi-threaded programs and consider the following types of events (other types can be easily added): write/read of variables, beginning/ending of function invocations, acquiring/releasing locks, and starts and exits of threads. A statement in the program may produce multiple events. Events

need to store enough information about the program state in order for the observer to perform its analysis.

Definition 32 *An **event** is a mapping of **attributes** into corresponding **values**. A **trace** is a sequences of events. We let τ, τ', etc., denote traces. From now on in the paper, we assume an arbitrary but fixed trace τ and let ξ denote ξ_τ (recall $\xi_\tau = \{e \mid e \in \tau\}$) for simplicity; events in ξ are called **concrete events**.*

For example, one event can be $e_1 : (counter = 8, thread = t_1,\ stmt = L_{11},\ type = write,\ target = a,\ state = 1)$, which is a write on location a with value 1, produced at statement L_{11} by thread t_1. One can easily include more information into an event by adding new attribute-value pairs. We use $key(e)$ to refer to the value of attribute *key* of event e. The attribute *state* contains the value associated to the event; specifically, for the write/read on a variable, $state(e)$ is the value written to/read from the variable; for ending of a function call, $state(e)$ is the return value if there is one; for the lock operation, $state(e)$ is the lock object; for other events, $state(e)$ is undefined. To distinguish among different occurrences of events with the same attribute values, we add a designated attribute to every event, *counter*, collecting the number of previous events with the same attribute-value pairs (other than the *counter*). This way, all events appearing in a trace can be assumed different.

6.3.2 Control Dependence on Events

Informally, if a change of $state(e)$ may affect the occurrence of e', then we say that e' has a *control dependence* on e, and write $e \sqsubset_{ctrl} e'$. For example, in Figure 6.3, the write on x at S_1 and the write on y at S_2 have a control dependence on the read on i at C_1, while the write on z at S_3 does not have such control dependence. Control dependence occurs inside of a thread, so we first define the total order within one thread:

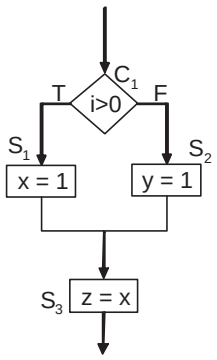

Figure 6.3: Control dependence

Definition 33 *Let $<$ denote the union of the total orders on events of each thread, i.e., $e < e'$ iff $thread(e) = thread(e')$ and $e <_\tau e'$.*

The control dependence among events in sliced causality is parametric in a control dependence relation among statements. In particular, one can use off-the-shelf algorithms for classic [42] or

for weak [74] control dependence. We chose to use the termination-sensitive control dependence (TSCD) introduced in [32] in our implementation of jPredictor[28]. Nevertheless, all we need to define sliced causality is a function returning the *control scope* of any statement C, say *scope* (C), which is the set of statements whose reachability depends upon the choice made at C, that is, the statements that control depend on C, for some appropriate or preferred notion of control dependence. Our approach also regards the lock acquire statement as a control statement that controls all the following statements, since the thread has to wait for the lock to continue its execution.

We assume that any control statement generates either a *read* event (the lock acquire is regarded as a read on the lock) or no event (the condition is a constant) when checking its condition. For the control statement with a complex condition, e.g., involving function calls and side effects, we can always transform the program to simplify its condition to a simple check of a boolean variable: one can compute the original condition before the control statement, store its result in a fresh boolean variable, and then modify the control statement to check only that variable in its condition.

Definition 34 *$e \sqsubset_{ctrl} e'$ iff $e < e'$, $stmt(e') \in scope(stmt(e))$, and e is "largest" with this property, i.e., there is no e'' such that $e < e'' < e'$ and $stmt(e') \in scope(stmt(e''))$.*

Intuitively, an event e is control dependent on the *latest* event issued by some statement upon which $stmt(e)$ depends. For example, in Figure 6.3, a write of x at S_1 is control dependent on the most recent read of i at C_1 and not on previous reads of i at C_1.

The *soundness* of analysis based on sliced causality is contingent to the *correctness* (no false negatives) of the employed control dependence: the analysis produces no false alarms when the *scope* function returns for each statement *at least* all the statements that control-depend on it. An extreme solution is to include all the statements in the program in each scope, in which case sliced causality becomes precisely the classic happen-before relation. As already pointed out in Section 2.2 and empirically shown in Section 7.2, such a choice significantly reduces the coverage of analysis. A better solution, still over-conservative, is to use weak dependence when calculating the control scopes. If termination information of loops is available, termination-sensitive control dependence can be utilized to provide correct and more precise results. One can also try to use the classic control dependence instead, but one should be aware that false bugs may be reported (e.g., when synchronization is implemented based on "infinite" loops).

6.3.3 Data Dependence on Events

If a change of $state(e)$ may affect $state(e')$ then we say e' has a *data dependence* on e and write $e \sqsubset_{data} e'$. Formally,

Definition 35 *For events e and e', $e \sqsubset_{data} e'$ iff $e <_\tau e'$ and one of the following holds:*

1. *$e < e'$, $type(e) = read$ and $stmt(e')$ uses $target(e)$ to compute $state(e')$;*
2. *$type(e) = write$, $type(e') = read$, $target(e) = target(e')$, and there is no other e'' with $e <_\tau e'' <_\tau e'$, $type(e'') = write$, and $target(e'') = target(e')$;*
3. *$e < e'$, $type(e') = read$, $stmt(e') \notin scope\ (stmt(e))$, and there exists some statement S in $scope\ (stmt(e))$ such that S can change the value of $target(e')$.*

The first case in this definition encodes the common data dependence. For example, for an assignment $x := E$, the write of x has data dependence on the reads generated by the evaluation of E. The second case in Definition 35 captures the interference dependence [62] in multithreaded programs, saying that a read depends on *the most recent* write of the same memory location. For instance, in Figure 6.3, if the observed execution is $C_1S_1S_3$ then the read of x at S_3 is data dependent on the most recent write of x at S_1. We treat lock release as a write on the lock and lock acquire as a read. The third case in Definition 35 is more intricate and relates to the relevant dependence in [45]. Assuming another execution of Figure 6.3, say $C_1S_2S_3$, no data dependence defined in cases *1* and *2* can be found in this run. However, the change of the value of the read of i at C_1 can potentially change the value of the read of x at S_3: if the value of i changes then C_1 may choose to execute the branch of S_1, resulting in a new write of x that may change the value of the read of x at S_3. Therefore, we say that the read of x at S_3 is data dependent on the read of i at C_1, as defined in case *3*. Note that although this dependence is caused by a control statement, it can *not* be caught by the control dependence; for example, the read of x at S_3 is *not* control dependent on the read of i at C_1 since $S_3 \notin scope(C_1)$. Aliasing information is needed to correctly compute dependence defined in case *3*, which one can obtain using any available techniques.

An important observation of Definition 35 is that there are no write-write, read-read, read-write data dependencies. Specifically, case *2* only considers the write-read data dependence, enforcing the read to depend upon only the latest write of the same variable. In other words, a write and the following reads of the same variable form an *atomic* block of events. This captures in a more general setting the work in [83].

6.3.4 Slicing Causality Using Relevance

When checking a trace τ against a property φ, not all the events in τ are relevant to φ; for example, to check dataraces on a variable x, accesses to other variables or function calls are irrelevant. Moreover, the *state* attributes of some relevant events may not be relevant; for example, the particular values written to or read from x for datarace (on x) detection. We next assume a generic *filtering function* that can be instantiated, usually automatically, to concrete filters depending upon the property φ under consideration:

Definition 36 *Let* α: $\mathsf{Events} \nrightarrow \mathsf{Events}$ *be a partial function, called a* ***filtering function****. The image of* α*, that is* $\alpha(\mathsf{Events})$*, is written more compactly* Events_α*; its elements are called* ***abstract relevant events****, or simply just* ***relevant events****. All thread start and exit events are relevant:* $\alpha(e)$ *defined whenever* $type(e) = start$ *or* $type(e) = exit$*.*

Let us assume an arbitrary but fixed property φ in what follows. Intuitively, $\alpha(e)$ is defined if and only if e is relevant to φ; if $\alpha(e)$ is defined, then $key(\alpha(e)) = key(e)$ for any attribute $key \neq state$, while $state(\alpha(e))$ is either undefined or equal to $state(e)$.

Definition 37 *Let* $\alpha(\tau)$*, written more compactly as* τ_α*, be the trace of relevant events achieved by applying* α *on events in* τ*. Let* ξ_α *denote* ξ_{τ_α} *for simplicity.*

This relevance-based abstraction plays a crucial role in increasing the predictive power of our analysis approach: in contrast to the concrete event set ξ, the corresponding abstract event set ξ_α allows more permutations of abstract events; instead of calculating permutations of ξ and then abstracting them into permutations of ξ_α like in traditional happen-before based approaches, we will calculate valid permutations of a *slice* of $\xi \cup \xi_\alpha$ that contains only events (directly or indirectly) relevant to φ. This slice is defined using the dependence on concrete and abstract events.

Definition 38 *All dependence relations are extended to abstract relevant events:*
If $e < / \sqsubset_{ctrl} / \sqsubset_{data} e'$ *then also* $\alpha(e) < / \sqsubset_{ctrl} / \sqsubset_{data} e'$*,* $e < / \sqsubset_{ctrl} / \sqsubset_{data} \alpha(e')$*,*
and $\alpha(e) < / \sqsubset_{ctrl} / \sqsubset_{data} \alpha(e')$*, whenever* $\alpha(e)$ *and/or* $\alpha(e')$ *is defined;*
$\sqsubset_{data}$ *is extended only when* $state(\alpha(e'))$ *is defined.*

We next define a novel dependence relation, called *relevance dependence*, which is concerned with *potential* occurrences of relevant events. Consider Figure 6.3 again. Suppose that relevant events include writes of y and z. For the execution $C_1S_1S_3$, only one relevant event is observed, namely the

write of z at S_3 (e'), which is not control dependent on the read of i generated at C_1 (e). Consider now another execution $C_1S_2S_3$; in addition to e', a new relevant event will be generated, namely the write of y at S_2, caused by the different choice made at C_1. Hence, a change of $state(e)$ *may affect the number of generated relevant events.* Formally, we define *relevance dependence* as follows:

Definition 39 *For $e \in \xi, e' \in \xi_\alpha$, we write $e \sqsubset_{rlvn} e'$ iff $e < e'$, $stmt(e') \notin scope(stmt(e))$, and there is a statement $S \in scope(stmt(e))$ that may generate a relevant event.*

Intuitively, if $e \sqsubset_{rlvn} e'$ then e' is not control dependent on e, but when $state(e)$ changes, some new relevant events may occur before e'. This may invalidate some permutations of ξ_α since valid permutations should preserve the *exact* number of relevant events.

Definition 40 *Let $\sqsubset$ be the relation $(\sqsubset_{data} \cup \sqsubset_{ctrl} \cup \sqsubset_{rlvn})^+$. If e and e' are concrete or relevant events such that $e \sqsubset e'$, then we say that e'* ***depends upon*** *e.*

Definition 41 *Let $\overline{\xi_\alpha} \subseteq \xi \cup \xi_\alpha$ be the* relevant slice *of events, extending ξ_α with events $e \in \xi$ such that $e \sqsubset e'$ for some $e' \in \xi_\alpha$. Let $\overline{\tau_\alpha}$ be the* ***abstract trace*** *of τ, i.e., the permutation of $\overline{\xi_\alpha}$ consistent with $<_\tau$.*

Intuitively, $\overline{\xi_\alpha}$ contains all the events that are directly or indirectly relevant to the property α. Our goal here is to define an appropriate notion of causal partial order on $\overline{\xi_\alpha}$ and then to show that any permutation consistent with it is sound. Recall that we fixed a trace τ; in what follows, τ' is used to refer to any arbitrary trace.

Definition 42 *Let $\prec^\tau \subseteq \overline{\xi_\alpha} \times \overline{\xi_\alpha}$ be the relation $(< \cup \sqsubset)^+$, which we call the* ***sliced causality*** *(or* ***sliced causal partial order****) of τ.*

From here on, by "causal partial order" we mean the sliced one. We next show that sliced causality is an instance of the parametric framework in Section 6.2.

Definition 43 *Let $\mathcal{C}_\alpha$: $\mathsf{Traces} \rightharpoonup \mathcal{PO}(\mathsf{Events})$ be the partial function defined as $\mathcal{C}_\alpha(\tau') = (\xi_{\tau'}, \prec^{\tau'})$ for each $\tau' \in \mathsf{Traces}$. Let $\mathcal{F}_\alpha \subseteq \mathsf{Traces}$ be the set of all possible abstract traces: for each $\tau_\mathcal{F} \in \mathcal{F}_\alpha$, there is some execution generating τ' such that $\overline{\tau'_\alpha} = \tau_\mathcal{F}$.*

We next give an informal proof of soundness of sliced causality. The proof is straightforward but a formal proof requires a formal semantics of the underlying programming language, like the one for Java defined in [40], which is beyond the scope of this thesis. Therefore, an informal description of the proof is given here, which can be easily expanded when the needed formalization is available.

In the rest of this section, we use ω, ω', ..., to denote fragments of traces and π, π', ..., to denote executions.

Definition 44 *A trace $\omega \in \mathcal{F}_\alpha$ is a **feasible prefix (of τ')** if and only if there exist an execution whose correspoing trace is τ' and also a trace $\omega' \in \mathcal{F}_\alpha$ such that $\overline{\tau'_\alpha} = \omega\omega'$.*

Intuitively, if ω is a feasible prefix then there exists an incomplete execution that generates ω.

Proposition 23 *If ωe and ω' are both feasible prefixes and for any $e' \prec^\omega e$, $e' \in \omega'$ then $\omega' e$ is also a feasible prefix.*

Proof:[skechy] Since $\omega e_1 e_2$ is a feasible prefix, there exists an execution π generating $\omega e_1 e_2$. From π we can construct another execution π' to generate $\omega e_2 e_1$ as follows. First, we follow the same execution as π from the beginning until ω is generated. Then we choose to continue on thread $thread(e_2)$ instead of $thread(e_1)$. By the formal semantics of the underlying programming language and based on by Definitions 34 and 35 (control- and date- dependence), we can prove that e_2 will be generated. Also, by Definition 39, we can prove that no new relevant event will be introduced before e_2. After e_2 is generated, we continue on thread $thread(e_1)$ and similarly, we can prove that e_1 will be generated without causing any new relevant event. This way, we can achieve a new execution generating $\omega e_2 e_1$. □

Corollary 10 *If ωe and ω' are both feasible prefixes and $\xi_\omega = \xi_{\omega'}$ then $\omega' e$ is also a feasible prefix.*

Proof: The result follows immediately from Proposition 23. □

Lemma 4 *If $\omega\omega''$ and ω' are both feasible prefixes and $\xi_\omega = \xi_{\omega'}$ then $\omega'\omega''$ is also a feasible prefix.*

Proof: It can be easily proved by induction on ω'' and by Corollary 10. □

Lemma 5 *If $\omega e_1 e_2$ is a feasible prefix and $e_1 \not\prec^\omega e_2$ then $\omega e_2 e_1$ is also a feasible prefix.*

Proof: Since $e_1 \not\prec^\omega e_2$, for any $e \prec^\omega e_2$, $e \in \xi_\omega$. By Proposition 23, ωe_2 is a feasible prefix. Also, for any $e \prec^\omega e_1$, $e \in \xi_\omega \subset \xi_{\omega e_2}$. Hence, by Proposition 23, $\omega e_2 e_1$ is a feasible prefix. □

Now we are ready to prove the main theorem:

Theorem 6 *$\mathcal{F}_\alpha$ is C-feasible, where C is C_α restricted to $\mathcal{F}_\alpha$. That is, for any abstract trace $\tau_\mathcal{F} \in \mathcal{F}_\alpha$, each liniearization of $\prec^{\tau_\mathcal{F}}$ corresponds to some possible execution of the multi-threaded system.*

Proof: Obviously, C_α restricted to $\mathcal{F}_\alpha$ is defined and total on $\mathcal{F}_\alpha$. For any $\tau' = \omega e_1 e_2 \omega' \in \mathcal{F}_\alpha$, if $C(\tau') = (\xi_{\tau'}, <)$ then $<=\prec^{\tau'}$. It is easy to prove that if $e_1 \not\prec^{\tau'} e_2$ then $e_1 \not\prec^{\omega e_1 e_2} e_2$. Since $\omega e_1 e_2$ is feasible prefix, by Lemma 5 $\omega e_2 e_1$ is also a feasible prefix. By Lemma 4, $\omega e_2 e_1 \omega' \in \mathcal{F}_\alpha$. The result then follows from Theorem 5. □

We can therefore analyze the permutations of relevant events consistent with sliced causality to detect potential violations *without* re-executing the program.

6.4 Predictive Runtime Analysis with Sliced Causality

Predictive runtime analysis with sliced causality is composed of two steps, namely, extracting sliced causality from the observed execution and checking the extracted sliced causality against desired property. For the former, we have developed an efficient algorithm using vector clocks and combined it with lock-atomicity to further increase the analysis coverage. For the latter, we have developed a width-first algorithm to generate all potential execution traces from the computed causality model, which can then be checked using the runtime morning technique discussed above. In Section 7.1, we also show that in practice, one may avoid the complexity of trace generation when focusing on generic concurent properties, like dataraces and atomicity violations.

6.4.1 Extracting Sliced Causality

We here describe a technique to extract from an execution trace of a multithreaded system the sliced causality relation corresponding to some property of interest φ. Our technique is *offline*, in the sense that it takes as input an already generated execution trace; that is because it needs to traverse the trace backwards. Our technique consists of two steps: ***(1)*** all the irrelevant events (those which are neither property events nor relevant events) are removed from the original trace, obtaining the *(φ)-sliced trace*; and ***(2)*** a *vector clock (VC)* based algorithm is applied on the sliced trace to capture the sliced causality partial order.

Slicing Traces

Our goal here is to take a trace ξ and a property φ, and to generate a trace ξ_φ obtained from ξ filtering out all its events which are irrelevant for φ. When slicing the execution trace, one must nevertheless keep all the property events. Moreover, one must also keep any event e with $e\ (\sqsubset_{ctrl} \cup \sqsubset_{data})^+\ e'$ for some property event e'. This can be easily achieved by traversing the

original trace backwards, starting with ξ_φ empty and accumulating in ξ_φ events that either are property events or have events depending on them already in ξ_φ. One can employ any off-the-shelf analysis tool for data- and control- dependence; e.g., our predictive analysis tool, jPredictor, uses termination-sensitive control dependence [32].

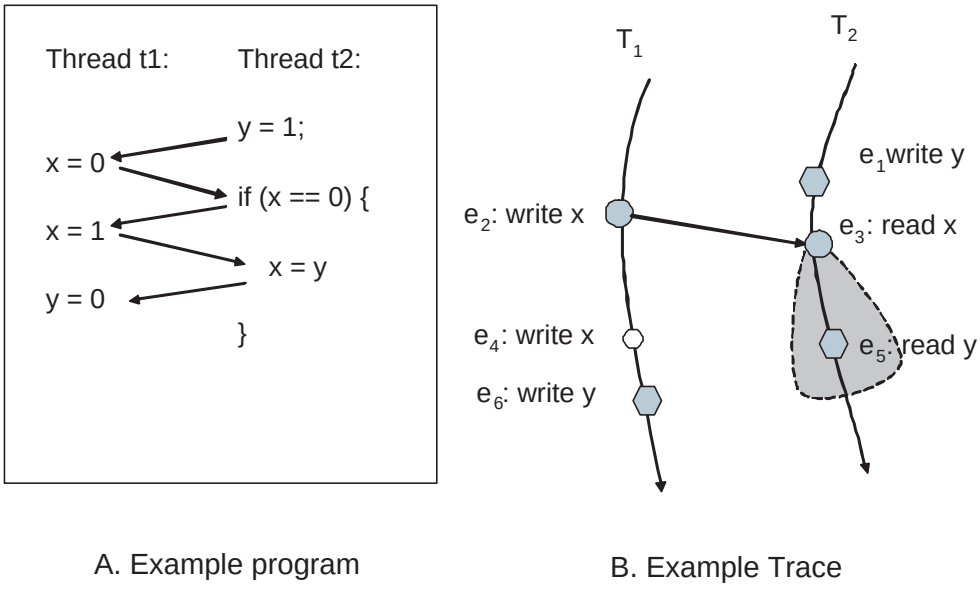

Figure 6.4: Example for relevance dependence

The algorithm informally described above is a variant of *dynamic program slicing* [2], where the slicing criterion is not the conventional reachability of a particular program statement, but, more generally, a set of event patterns determined by the desired property (e.g., reads/writes of a shared location for dataraces, etc.). Unfortunately, one backwards traversal of the trace does not suffice to correctly calculate all the relevant events. Let us consider the example in Figure 6.4. When the backward traversal first reaches e_4, it is unclear whether e_4 is relevant or not, because we have not seen e_3 and e_2 yet. Thus a second scan of the trace is needed to include e_4. Once e_4 is included in ξ_φ, it may induce other relevance dependencies, requiring more traversals of the trace to include them. This process ceases only when no new relevant events are detected and thus resulting sliced trace stabilizes.

As the discussion preceding Theorem 6 shows, if one misses relevant events like e_4 then one may "slice the trace too much" and, consequently, one may produce false alarms. Because at each trace traversal some event is added to ξ_φ, the worse-case complexity of the sound trace slicing procedure is square in the number of events. Since execution traces can be huge, in the order of billions of events,

any trace slicing algorithms that is worse than linear may easily become prohibitive. For that reason, jPredictor slices the trace only once, thus achieving an approximation of the complete slice that can, in theory, lead to false alarms. However, our experiments show that this approximation is actually very precise in practice: all the programs that we have evaluated follow our approximation (Section 7.2).

Capturing Sliced Causality with Vector Clocks

Vector clocks [63] are routinely used to capture causal partial orders in distributed and concurrent systems. A *VC*-based algorithm was presented in [82] to encode a conventional multithreaded-system "happen-before" causal partial order on the unsliced trace. We next adapt that algorithm to work on our sliced trace and thus to capture the sliced causality. Recall from [82] that a vector clock (VC) is a function from threads to integers, $VC : T \rightarrow Int$. We say that $VC \leq VC'$ iff $\forall t \in T$, $VC(t) \leq VC'(t)$. The max function on *VCs* is defined as: $\max(VC_1, ..., VC_n)(t) = \max(VC_1(t), ..., VC_n(t))$.

Before we explain our *VC* algorithm, let us introduce our event and trace notation. An *event* is a mapping of *attributes* into corresponding *values*. One event can be, e.g., $e_1 : (counter = 8, thread = t_1,\ stmt = L_{11},\ type = write,\ target = a,\ state = 1)$, which is a write on location a with value 1, produced at statement L_{11} by thread t_1. One can include more information into an event by adding new attribute-value pairs. We use $key(e)$ to refer to the value of attribute *key* of event e. To distinguish different occurrences of events with the same attribute values, we add a designated attribute to every event, *counter*, collecting the number of previous events with the same attribute-value pairs (other than the *counter*). A *trace* is a finite sequence of events. From here on, our default trace is the φ-sliced trace ξ_φ obtained in Section 6.4.1.

Intuitively, vector clocks are used to track and transmit the causal partial ordering information in a concurrent computation, and are typically associated with elements participating in such computations, such as threads, processes, shared variables, messages, signals, etc. If *VC* and *VC'* are vector clocks such that $VC(t) \leq VC'(t)$ for some thread t, then we can say that *VC'* has newer information about t than *VC*. In our *VC* technique, every thread t keeps a vector clock, VC_t, maintaining information about all the threads obtained both locally and from thread communications (reads/writes of shared variables). Every shared variable is associated with two vector clocks, one for writes (VC_x^w) used to enforce the order among writes of x, and one for all accesses (VC_x^a) used to accumulate information about all accesses of x. They are then used together to keep the order between writes and reads of x. Every property event e found in the analysis is associated a *VC*

attribute, which represents the computed causal partial order. We next show how to update these VCs when an event e is encountered (the third case can overlap the first two; if so, the third case will be handled first):

1. $type(e) = write$, $target(e) = x$, $thread(e) = t$ (the variable x is written in thread t) and x is a shared variable. In this case, the write vector clock VC_x^w is updated to reflect the newly obtained information; since a write is also an access, the access VC of x is also updated; we also want to capture that t committed a causally irreversible action, by updating its VC as well: $VC_t \leftarrow VC_x^a \leftarrow VC_x^w \leftarrow \max(VC_x^a, VC_t)$.
2. $type(e) = read$, $target(e) = x$, $thread(e) = t$ (the variable x is read in t), and x is a shared variable. Then the thread updates its information with the write information of x (we do not want to causally order reads of shared variables!), and x updates its access information with that of the thread: $VC_t \leftarrow \max(VC_x^w, VC_t)$ and then $VC_x^a \leftarrow \max(VC_a^x, VC_t)$.
3. e is a property event and $thread(e) = t$. In this case, let $VC(e) := VC_t$. Then $VC_t(t)$ is increased to capture the intra-thread total ordering: $VC_t(t) \leftarrow VC_t(t) + 1$.

The vector clocks associated with property events as above soundly, but incompletely, capture the sliced causality:

Theorem 7 *$e \prec e'$ implies $VC(e) \leq VC(e')$.*

The proof of Theorem 7 can be (non-trivially) derived from the one in [82]. The extension here is that the dependence is taken into account when computing the sliced trace. Note that, unlike in [82], the partial order $\leq$ among *VCs* is *stronger* than the causality. This is because when *VCs* are computed, the write-after-read order is also taken into account (the first case above), which the sliced causality $\prec$ does not need to encode. We do not know how to faithfully capture the sliced causality using *VCs* yet. Nevertheless, soundness is not affected because Theorems 6 and 7 yield the following:

Corollary 11 *Any permutation of property events consistent with $\leq$ (on events' VCs) is sound w.r.t. the sliced causality $\prec$.*

6.4.2 Causality with Lock-Atomicity

Happen-before techniques for detecting/predicting concurrency bugs typically rely on checking, directly or indirectly, linearizations of events consistent with a causal partial order. Two happen-before

techniques that we are aware of generalize the concept of causality beyond a partial-order. One example is [72], which proposes a hybrid approach combining the happen-before causality with lock-set techniques for detecting data-races. The algorithm in [72] is, unfortunately, unsound and specialized for detecting data-races, so it cannot be used to generate sound linearizations of events to check against arbitrary properties. Another example is [83], which groups a write atomicity with all its subsequent reads; the resulting causality is extended with atomicity information and new linearizations of events are allowed, namely those that permute groups of events in the same atomic block. In this subsection we present another generalization of causality beyond a partial-order, one borrowing the idea of atomic blocks from the technique in [83], but whose atomicity is given by the semantics of locks rather than by writes and subsequent reads of shared variables. Unlike the technique in [72], our novel causality is general purpose, in the sense that it still allows for sound linearizations of events, so it can be used in combination with any trace property, data-races being only a special case. Even though in this paper we use this improved causality as a generalization of our sliced causality, the idea is general and can be used in combination with any other happen-before causality.

Our causality with lock-atomicity below is reminiscent of the notion of synchronization dependence defined in [46] and used for static slicing there. However, our causality is based on runtime events instead of statements in the control-flow graph, and is used to assure the (dynamic) causal atomicity of lock-protected blocks.

Thread t_1:

e_{11}(type = read, target = y ...)
e_{12}(type = write, target = y ...)
e_{13}(type = acquire, target = lock ...)
e_{14}(type = read, target = x ...)
e_{15}(type = write, target = x ...)
e_{16}(type = release, target = lock ...)

Figure 6.5: Event trace containing lock operations

A simple sound approach to partially incorporating lock semantics into causality, followed for example in [82], is to regard locks as shared variables and their acquire and release as reads and writes. This way, blocks protected by the same lock are ordered and kept separate. However, this ordering is stronger than the actual lock semantics (which only requires mutual exclusion). We extend our sliced causality to take into account the actual semantics of lock-atomicity: the set of events generated within a lock-protected block are considered lock-atomic. Two lock-atomic event sets w.r.t. the same lock cannot be interleaved, but *can be permuted* if there are no other causal constraints on them. Consider two more types of events for lock operations, *acquire* and *release*, whose target is the accessed lock. If there are embedded

same-lock lock operations (a thread can acquire the same lock multiple times), only the outermost acquire-release event pair is considered. Figure 6.5 shows a trace containing lock events.

From here on, let τ be any execution trace. Let "$<$" be the union of the total intra-thread orders induced by τ, that is, $e < e'$ iff $\mathit{thread}(e) = \mathit{thread}(e')$ and e appears before e' in τ. Let $\prec$ be any partial order on the events in τ. The lock-atomicity technique described below can be therefore used in combination with any (causal) partial order. We are, however, going to apply the subsequent results for a particular causality, namely the sliced causality restricted to relevant and property events (see Corollary 12). When doing that, we also discard all those irrelevant acquire/release events (i.e., those synchronizing only irrelevant events).

Definition 45 *Events e_1 and e_2 are* l-atomic*, written $e_1 \Updownarrow_l e_2$, if and only if there is some event e such that $\mathit{type}(e) = \mathit{acquire}$, $\mathit{target}(e) = l$, $e < e_1$, $e < e_2$, and there is no e' with $\mathit{type}(e') = \mathit{release}$, $\mathit{target}(e) = l$, and $e < e' < e_1$ or $e < e' < e_2$. For each lock l, we let $[e]_l$ denote the* l-atomic equivalence class of e.

In Figure 6.5, $e_{14} \Updownarrow_{lock} e_{15}$. We capture the lock-atomicity as follows. A counter c_l is associated with every lock l. Each thread stores the set of locks that it holds in LS_t. Events are enriched with a new attribute, LS, which is a partial mapping from locks into corresponding counters. When event e is processed, the lock information is updated as follows:

1. if $\mathit{type}(e) = \mathit{acquire}$, $\mathit{thread}(e) = t$, and $\mathit{target}(e) = l$, then $c_l = c_l + 1$, $LS_t = LS_t \cup \{l\}$;
2. if $\mathit{type}(e) = \mathit{release}$, $\mathit{thread}(e) = t$, and $\mathit{target}(e) = l$, then $LS_t = LS_t - \{l\}$.
3. $LS(e)(l) = c_l$ for all $l \in LS_{\mathit{thread}(e)}$ and $LS(e)(l)$ undefined for any other l;

If we write $LS(e)(l) = LS(e')(l)$ then we mean that the two are *defined and equal*. The following important result holds:

Theorem 8 *$e \Updownarrow_l e'$ iff $LS(e)(l) = LS(e')(l)$.*

Intuitively, a permutation of events is consistent, (or sound, or realizable, or possible during an execution) if and only if it preserves *both* the sliced causality and the lock-atomicity relation above.

Definition 46 *A **cut** Σ is a set of events in τ. Σ is **consistent** if and only if for all $e, e' \in \tau$,*

(a) if $e' \in \Sigma$ and $e \prec e'$ then $e \in \Sigma$; and

(b) if $e, e' \in \Sigma$ and $e' \notin [e]_l$ for a lock l, then $[e']_l \subseteq \Sigma$ or $[e]_l \subseteq \Sigma$.

The first item says that for any event in Σ, all the events upon which it depends should also be in Σ. The second property states that there is *at most one* incomplete lock-atomic set for every particular lock l in Σ. Otherwise, the lock-atomicity is broken. Essentially, Σ contains the events in the prefix of a consistent permutation. When an event e can be added to Σ without breaking the consistency, e is called *enabled* for Σ.

Definition 47 *Event $e' \in \tau - \Sigma$ is **enabled** for consistent cut Σ iff*

(a) for any event $e \in \tau$, if $e \prec e'$ then $e \in \Sigma$; and

(b) for any $e \in \Sigma$ and any lock l, either $e' \in [e]_l$ or $[e]_l \subseteq \Sigma$.

Hence, e is enabled for consistent cut Σ iff $\Sigma \cup \{e\}$ is also consistent.

Definition 48 *A **consistent permutation** $e_1 e_2 ... e_{|\tau|}$ of τ is one that generates a sequence of consistent cuts $\Sigma_0 \Sigma_1 ... \Sigma_{|\tau|}$: for all $1 \leq r \leq |\tau|$, Σ_{r-1} is consistent, e_r is enabled for Σ_{r-1}, and $\Sigma_r = \Sigma_{r-1} \cup \{e_r\}$.*

The following result holds and shows the soundness of lock-atomicity:

Theorem 9 *Any consistent permutation of τ corresponds to some possible execution of the multi-threaded system.*

The proof is non-trivial and can be done by extending the proof of Theorem 6 (using the parametric framework for causality like in [30]) to incorporate lock-atomicity.

Corollary 12 *Consider now our original trace ξ together with its φ-sliced trace ξ_φ. Then any permutation of property events that is consistent with the sliced causality and the lock-atomicity corresponds to some possible execution of the multi-threaded system.*

6.4.3 Generating Potential Runs

We here discuss an algorithm to check all the consistent permutations of events against the desired property φ. The actual permutations of events are *not* generated, because that would be prohibitive. Instead, a monitor is assumed for the property φ which is run synchronously with the generation of the next level in the computation lattice, following a breadth-first strategy. Figure 6.6 gives a high-level pseudocode to generate and verify, on a level-by-level basis, potential runs consistent with the sliced causality with lock-atomicity. ξ_φ is the set of relevant events. *CurrentLevel* and *NextLevel* are sets of cuts. We encode cuts Σ as: a $VC(\Sigma)$ which is the max of the *VCs* of all its threads

```
globals ξ_φ ← φ-sliced trace, CurrentLevel ← {Σ_{0...0}}
procedure main()
  while (ξ_φ ≠ ∅) do verifyNextLevel()
endprocedure
procedure verifyNextLevel()
  local NextLevel ← ∅
  for all e ∈ ξ_φ and Σ ∈ CurrentLevel do
    if enabled(Σ, e) then NextLevel ← NextLevel ∪ createCut(Σ, e)
  CurrentLevel ← NextLevel
  ξ_φ ← removeRedundantEvents()
endprocedure
procedure enabled(Σ, e)
  return VC(e)(thread(e)) = VC(Σ)(thread(e)) + 1   and
         VC(e)(t) ≤ VC(Σ)(t) for all t ≠ thread(e)   and
         LS(e)(l) = LS(Σ)(l) when both defined, for all locks l
endprocedure
procedure createCut(Σ, e)
  Σ' ← new copy of Σ
  VC(Σ')(thread(e)) ← VC(Σ)(thread(e)) + 1
  if type(e) = acquire and target(e) = l then LS(Σ')(l) ← LS(e)(l)
  if type(e) = release and target(e) = l then LS(Σ')(l) ← undefined
  MS(Σ') ← runMonitor(MS(Σ), e)
  if MS(Σ') = "error" then reportViolation(Σ, e)
  return Σ'
endprocedure
```

Figure 6.6: Consistent runs generation algorithm

(updated as shown in procedure *createCut*); a partial mapping $LS(\Sigma)$ which keeps for each lock l its current counter c_l (updated as also shown in *createCut*); and the current state of the property monitor for this run, *MS*. The property monitor can be any program, in particular those generated automatically from specifications, like in MOP [29].

Figure 6.7 shows a simple example for generating consistent permutations. Figure 6.7 (A) is an observed execution of a two-thread program. The solid arrow lines are threads, the dotted arrows are dependencies, and the dotted boxes show the scopes of synchronized blocks. Both synchronized blocks are protected by the same lock, and all events marked here are relevant. Figure 6.7 (B) illustrates a run of the algorithm in Figure 6.6, where each level corresponds to a set of cuts generated by the algorithm. The labels of transitions between cuts give the added events. Initially, there is only one cut, Σ_{00}, on Level 0. The algorithm first checks every event in ξ_φ and every cut in the current level to generate cuts of the next level by appending enabled events to current cuts. The *enabled* procedure implements the definition of a consistent permutation (compares the *VC*s between a candidate event and a cut, then checks for the compatibility of their lock-atomicity). For example, only e_{11} is enabled for the initial cut, Σ_{00}; e_{12} is enabled for Σ_{10} on Level 1, but e_{21} is not because of

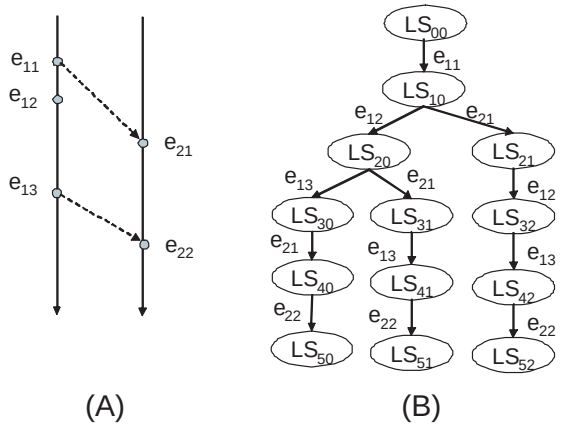

Figure 6.7: Example for consistent run generation

the lock-atomicity. On Level 2, after e_{11} and e_{12} have been consumed, e_{21} and e_{13} are both enabled for the cut Σ_{20}. If an event e is enabled for a cut Σ, e is added to Σ to create a new cut Σ', as depicted by the transitions in Figure 6.7. The vector clocks and lock set information of Σ' will be computed according to e. After the next level is generated, redundant events, e.g., e_{11} after Level 1, will be removed from ξ_φ. Also, the property monitor in Σ will be run (see the *createCut* procedure) and its new state stored in Σ'; violations are reported as soon as detected.

The pseudocode in Figure 6.6 glossed over many implementation details that make it efficient. For example, ξ_φ can be stored as a set of lists, each corresponding to a thread. Then the *VC* of a cut Σ can be seen as a set of pointers into each of these lists. The potential event e for the loop in *verifyNextLevel* can only be among the next events in these lists. The function *removeRedundantEvents()* eliminates events at the beginning of these lists when their *VC*s are found to be smaller than or equal to the *VC*s of all the cuts in the current level. In other words, to process an event, a good implementation of the algorithm in Figure 6.6 would take time $O(|\mathit{Threads}|)$.

Chapter 7

jPredictor

The proposed predictive runtime analysis technique have been implemented in a tool for multi-threaded Java programs, called jPredictor. We have applied jPredictor to a comprehensive set of real-world applications to evaluate its effectiveness in practice. The results are encouraging: after the programs under analysis were executed only once, jPredictor found all the errors reported by other tools; it also found errors missed by other tools, including static race detectors, as well as unknown errors in popular systems like Tomcat and the Apache FTP server.

7.1 Implementation

jPredictor is a runtime analysis tool to detect concurrent bugs in Java programs using sliced causality with lock-atomicity. In addition to an efficient implementation of the vector-clock-based algorithm discussed above, jPredictor also provides an optimal instrumentation framework to log and replay program execution, as well as specialized property checkers for data races and atomicity. Interested readers can find more information on jPredictor at its website [57], where it is also available for download.

7.1.1 Architecture

jPredictor is composed of two major components: the program instrumentor and the trace predictor (Figure 7.1). The program instrumentor instruments the program under testing with instructions that log the execution. To reduce the runtime overhead caused by monitoring, only partial information is logged during execution. The trace predictor analyzes the logged execution trace to predict potential bugs using sliced causality. If a possible bug is detected, jPredictor generates an abstract execution trace leading to it, which explains how the bug can be hit in a real execution.

As shown in Figure 7.2, the trace predictor consists of four stages: the pre-processor, the trace slicer, the *VC* calculator, and the property checker. The role of the pre-processor is two-fold. First,

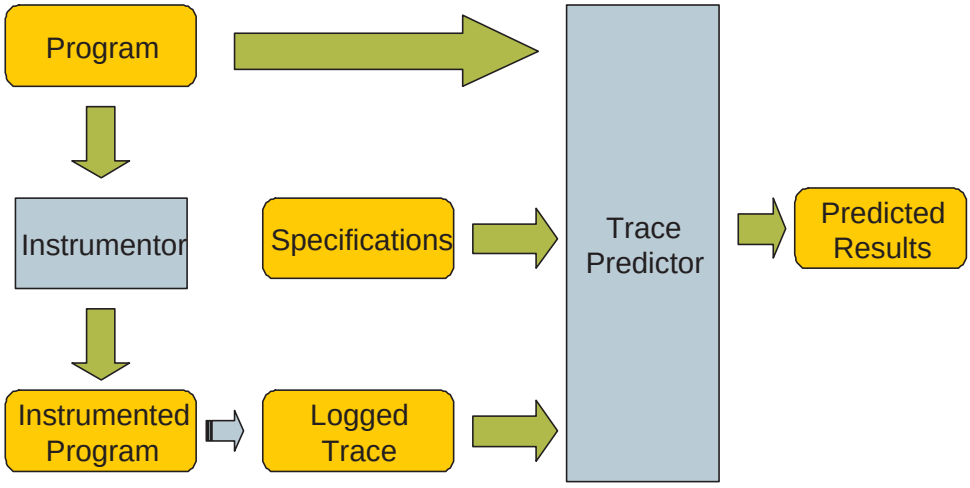

Figure 7.1: Working Architecture of jPredictor

it constructs a more informative trace from the partially logged trace using static analysis on the original program, providing a foundation for the subsequent analysis. Second, it identifies all the shared locations in the observed execution, which are critical for a precise predictive analysis. The slicer scans the re-constructed trace, producing a trace slice for every property to check. The generated slices are fed into the VC calculator, which computes the sliced causality as discussed in Section 6.4.1. In the last stage, the property checker verifies the execution against the desired property using the computed sliced causal with lock-atomicity. All these stages communicate using plain ASCII text, making the tool easy to extend. For example, we also implemented a conventional happen-before slicer to generate trace slices containing all the shared variable accesses. This way, we were able to compute the traditional happen-before causality without changing any other components of jPredictor; it has been used in [30] for comparison purposes. We next give more details about each component of jPredictor.

7.1.2 Partial Monitoring

Program monitoring plays a fundamental role in predictive runtime analysis. For sliced causality, *complete* monitoring, i.e., observing every instruction of the program, is desired for an accurate data-flow analysis. However, such monitoring imposes huge runtime overhead that we want to avoid in practice. An important observation here is that, by using static analysis, one can replay the execution *with much less observation* of the program. The more complicated the static analysis, the fewer observation points and the less monitoring overhead one can achieve. For example, one could

symbolically execute the program and only need runtime information when the symbolic execution cannot decide how to proceed. How to achieve minimum but sufficient information by runtime monitoring in order to replay an execution is an interesting question by itself, but out of the scope of this paper. In what follows, we briefly discuss an effective solution adopted by jPredictor, which aims at reducing the monitoring overhead with relatively simple static analysis.

Two components of jPredictor are involved in obtaining a complete trace via partial monitoring, namely the program instrumentor and the pre-processor of the trace predictor (Figures 7.1 and 7.2). The program instrumentor, built on top of Soot [85], a Java bytecode engineering package, is used to insert logging instructions into the original program. Three kinds of program points are observed: beginnings of methods, targets of conditionals, and accesses to objects/arrays (i.e., field/element access or method invocations). To faithfully replay an execution, we need to know which method implementation was actually executed when a method invocation is encountered. Because of the polymorphism and the virtual method mechanism of Java, it can be difficult to identify the actual target method implementation at a specific program point by static analysis, while logging the entry of the method at runtime is a simple and precise solution. Similarly, static conditional analysis is often difficult and imprecise but can be totally avoided using runtime information. For object accesses, static aliasing analysis is usually expensive and often imprecise in the absence of runtime information. Admittedly, monitoring every object/array access is fairly heavy and can be improved by advanced aliasing analysis, but we leave that to future research; the current solution yielded reasonable runtime overhead in our experiments.

The information logged for every event is as follows. Each event should carry along the id of the thread which issued it. In addition, the begin method event records a full signature of the method, so that we can locate the actual method implementation later; the branch target event needs to store the line number of the target; the object/array access event contains the id of the accessed object/array and, for the array access, the index of the element. Instead of analyzing this partial trace directly to compute the sliced causality, jPredictor first re-constructs a complete trace out of it using the pre-processor. The constructed complete trace can be reused by different trace slicers to verify different properties. Also, the trace slicer becomes simpler and independent of the instrumentation strategy. The pre-processor also works at the Java bytecode level. It goes through the instructions of the program following the control flow information recorded in the logged trace, supplementing the trace with useful information ignored by the monitoring. More specifically, the following information is added into the trace: return points of method executions, origins of jump

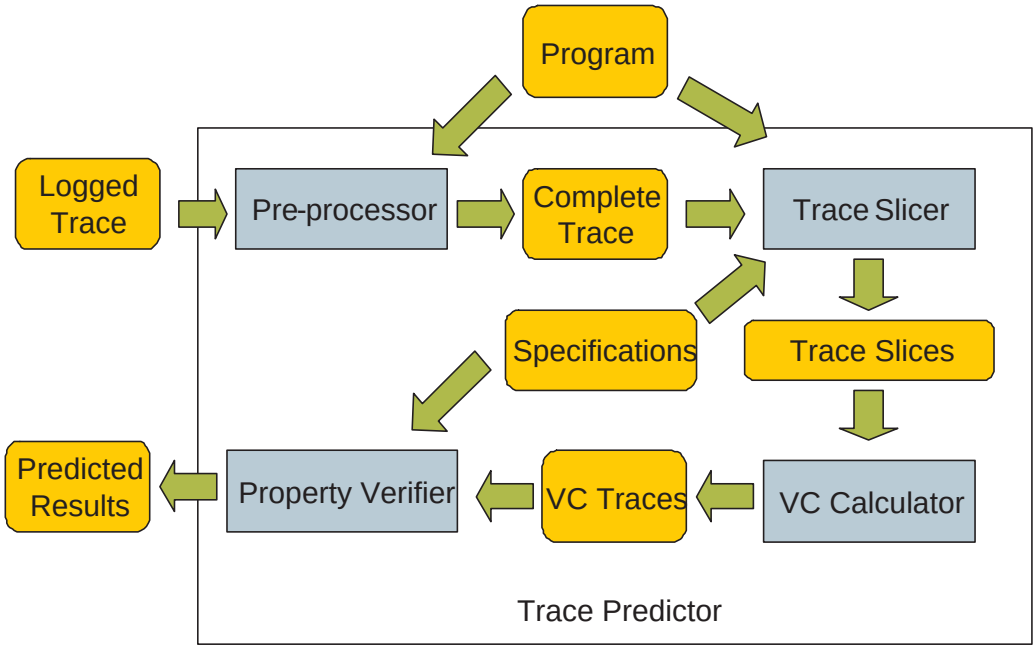

Figure 7.2: Staged Architecture of Trace Predictor

statements, field names of object accesses, starts of threads, and lock/unlock events. The first two are needed because the slicing is performed in a backward manner; field names are used to achieved fine grained data analysis; thread start events and lock-related events are needed to compute correct VCs and lock-atomicity.

It is impossible or undesired to instrument all the methods used in a program, e.g., the (native) Java library functions. jPredictor by default does not instrument any Java library function unless requested by the user. However, without further knowledge, the data-flow dependence analysis of jPredictor may lose information and produce imprecise results. For example, when the *ArrayList.addAll(Collection c)* is called, the target *ArrayList* object will be changed according to the input argument *c*. A conservative solution is to assume all the instrumented methods *impure*, i.e., they may change the target object and any of the input arguments. This assumption is often too restrictive, because many functions simply return the state of the object. Hence, jPredictor allows the user to input purity information about functions that are used but not instrumented in the program. If no information is provided for a certain function, the function will be considered impure. The purity information can be reused for different programs and we have pre-defined the purity of many Java library functions; that was sufficient to do our experiments entirely automatically.

7.1.3 Trace Slicing and VC Calculator

The trace slicer implements the dynamic slicing algorithm described in Section 6.4.1 to extract property-specific trace slices from the completed trace. jPredictor handles all thread-related, e.g., *start* and *join*, and all synchronization events specially, according to the Java semantics. Also, for efficiency purposes, only one pass of the backward slicing is performed in the present implementation. This may result in a trace slice that does not contain all the relevant events and thus lose the soundness of the predictive analysis. In other words, the current jPredictor prototype may produce false alarms due to incomplete trace slices. However, this deliberately unsound implementation proved to be sufficiently effective in our evaluation: no false alarm has been reported in our experiments.

The trace slice is then used by the *VC* calculator to compute *VC*s based on the algorithm described in Section 6.4.1. Similarly to the slicer, Java-specific language features are specially handled during the *VC* computation to ensure the correctness and accuracy of the results, for example, the beginning of a thread execution should depend on the corresponding thread creation event. The output of the *VC* calculator is a sequence of property events associated with *VC*s and lock-atomicity information. This output is then verified against the desired property as explained below.

7.1.4 Verifying Properties

The property checker of jPredictor implements the algorithm in Figure 6.7 to generate consistent permutations of property events, providing a generic way to verify temporal properties against the observed execution using the sliced causality. In other words, jPredictor is not bound to any particular type of property: one can hook up any property monitor to jPredictor to predict possible violations of the desired property which can be specified using any trace specification language, e.g., regular expressions, temporal properties, or context-free languages. This way, combining it with the automatic monitor generation framework provided by JavaMOP [29], jPredictor gives an automated platform to formally specify and dynamically verify trace properties against concurrent Java programs.

Unfortunately, generating all the consistent permutations of a partial order is a #P-complete problem [23] and can be unnecessarily expensive for those properties for which we can have more efficient solutions. jPredictorprovides two specialized checkers to detect data races and atomicity violations efficiently using sliced causality and lock-atomicity. We next discuss both of them briefly.

First, we need to determine the property events for these two types of properties: the property events for detecting races of a specific memory location (i.e., the same object field or array element) are all the writes/reads of the memory location; the property events for analyzing the atomicity of atomic blocks are all the accesses of shared locations used within those blocks. Let $\prec^x_{race}$ be the sliced causality for detecting the data race of the shared location x. We then formally define a data race as follows:

Definition 49 *For two events, e_1 and e_2, if they access the same memory location x and at least one of them is a write, then we say that they cause a* data race *on x iff e_1 and e_2 are not comparable under $\prec^x_{race}$ and they are not protected by the same lock.*

Therefore, the data race can be detected by comparing events generated in different threads. In our implementation, the property events, writes/reads of the shared variable in this case, are processed following the order in the logged trace (i.e., the order in the original execution). When a new property event is processed, it is checked against those events processed in other threads using the above race condition. The complexity of this algorithm is square to the number of property events.

Thanks to the genericity of JPREDICTOR with regards to properties, one is allowed to use existing algorithms, e.g., the reduction based algorithm in [43] or the causality based algorithm in [41], to check the atomicity on the consistent permutations generated by JPREDICTOR. The specialized efficient algorithm implemented in JPREDICTOR to identify atomicity violations is based on the problematic scenarios proposed in [87]. In short, JPREDICTOR first constructs all the atomic blocks from the execution trace; then each pair of blocks generated in different threads is examined to see if any of the 11 violation patterns in [87] can be matched under the sliced causality and lock-atomicity constraints. Since those patterns involve at most two different variables, the complexity of checking each pair is $O(m^2)$, where m is the number of events in both blocks. The worse case complexity of this atomicity algorithm is therefore $O(n^4)$, where n is the number of all the property events.

When both data races and atomicity are checked, the analysis cost can be further reduced by reusing the trace slices generated for race detection in the atomicity analysis. More specifically, if an atomic block B contains accesses of shared variables $x_1, ...x_i$, the trace slice for checking the atomicity of B is the union of the slices for checking races on $x_1, ..., x_i$. The formal proof of the correctness is left out of this paper. Intuitively, for two sets of events, E_1 and E_2, if trace slices ξ_1 and ξ_2 contain all the events affecting E_1 and E_2, respectively, then $\xi_1 \cup \xi_2$ should also contain all the events affecting $E_1 \cup E_2$. This way, one does not need to re-slice the trace for atomicity analysis if

Program	LOC	Threads	S.V.	Slowdown
Banking	150	3	10	0.34
Elevator	530	4	123	N/A
tsp	706	4	648	7.05
sor	17.7k	4	102	0.47
hedc	39.9k	10	119	0.56
StringBuffer	1.4k	3	7	0.61
Vector	12.1k	18	49	0.79
IBM Web Crawler	unknown	7	76	0.01
StaticBucketMap	748	6	381	35.6
Pool 1.2	5.8k	2	119	0.29
Pool 1.3	7.0k	2	95	0.32
Apache Ftp Server	22.0k	12	281	N/A
Tomcat Components	4.0k	3	13	0.1

Table 7.1: Benchmarks

race detection has been carried out. As our experiments show, merging trace slices is much cheaper than generating slices (Section 7.2.3). So the analysis performance can be significantly improved by reusing existing slices.

7.2 Evaluation

Here we present evaluation results of jPredictor on two types of common and well-understood concurrency properties, which need no formal specifications to be given by the user and whose violation detection is highly desirable: dataraces and atomicity. jPredictor has also been tried on properties specified formally and monitors generated using the MOP [29] logic-plugins, but we do not discuss those here; the interested reader is referred to [28]. We discuss some case studies, showing empirically that the proposed predictive runtime verification technique is viable and that the use of sliced causality significantly increases the predictive capabilities of the technique. All experiments were performed on a 2.6GHz X2 AMD machine with 2GB memory. Interested readers can find detailed result reports on jPredictor's website at [57].

7.2.1 Benchmarks

Table 7.1 shows the benchmarks that we used, along with their size (lines of code),[1] number of threads created during their execution, number of shared variables (S.V.) detected, and slowdown ratios after instrumentation [2]. Banking is a simple example taken over from [39], showing relatively classical concurrent bug patterns. Elevator, tsp, sor and hedc come from [88]. Elevator is a discrete event simulator of an elevator system. tsp is a parallelized solution to the traveling salesman problem. sor is a scientific computation application synchronized by barriers instead of locks. hedc is an application developed at ETH that implements a meta-crawler for searching multiple Internet achieves concurrently.

StringBuffer and Vector are standard library classes of Java 1.4.2 [52]. IBM web crawler is a component of the IBM Websphere tested in [37]. [3] StaticBucketMap, Pool 1.2 and 1.3 are part of the Apache Commons project [5]: StaticBucketMap is a thread-safe implementation of the Java Map interface; Pool 1.2 and 1.3 are two versions of the Apache Commons object pooling components. Apache FTP server [6] is a pure Java FTP server designed to be a complete and portable FTP server engine solution. Tomcat [86] is a popular open source Java application server. The version used in our experiments is 5.0.28. Tomcat is so large, concurrent, and has so many components, that it provides a base for almost unlimited experimentation all by itself. We only tested a few components of Tomcat, including the class loaders and logging handlers.

For most programs, we used the test cases contained in the original packages. The Apache Commons benchmarks, i.e., StaticBucketMap and Pool 1.2/1.3, provide no concurrent test drivers, but only sequential unit tests. We manually translated some of these into concurrent tests by executing the tests concurrently and modifying the initialization part of each unit test method to use a shared global instance. For StringBuffer and Vector, some simple test drivers were implemented, which simply start several threads at the same time to invoke different methods on a shared global object. The present implementation of JPREDICTOR tracks accesses of array elements, leading to the large numbers of shared variables and significant runtime overhead in tsp and StaticBucketMap. For other programs, the runtime overhead is quite acceptable.

Each test was executed *precisely once* and the resulting trace has been analyzed. While multiple

[1] Different papers give different numbers of lines of code for the same program due to different settings. In our experiments, we counted those files that were instrumented during the testing, which can be more than the program itself. For example, the kernel of hedc contains around 2k lines of code; but some other classes used in the program were also instrumented and checked, e.g., a computing library developed at ETH. This gave us a much larger benchmark than the original hedc.

[2] Not applicable for some programs, e.g., Elevator.

[3] No source code is available for this program.

Program	Var	Trace Size		Running Time (seconds) per S.V.				Races		
		Logged	Complete	Preprocess	Slice	VC	Verify	Harmful	Benign	False
Banking	10	244	320	0.01	0.04	0.01	0.01	1	0	0
Elevator	48	62314	71269	1.0	8.1	1.2	0.15	0	0	0
tsp	47	141239	237801	2.2	26.1	2.3	0.23	1	0	0
sor	17	10968	12654	0.3	1.9	0.2	0.01	0	0	0
hedc	43	128289	183317	2.1	17.9	0.16	0.01	4	0	0
StringBuffer	4	738	871	0.06	0.28	0.05	0.01	0	0	0
Vector	47	876	1086	0.08	0.3	0.06	0.01	0	1	0
IBM Web Crawler	59	3128	3472	0.18	0.6	0.16	0.01	1	3	0
StaticBucketMap	39	319482	366743	7.6	131.6	12.2	0.03	1	0	0
Pool 1.2	54	20541	24072	0.26	1.42	0.34	0.01	35	0	0
Pool 1.3	45	1426	1669	0.16	0.76	0.23	0.01	1	0	0
Apache FTP Sever	71	19765	20047	0.69	3.87	0.34	0.02	11	5	0
Tomcat Components	13	3240	3698	0.21	0.62	0.2	0.01	2	2	0

Table 7.2: Race detection results. Var: variables to check. S.V.: Shared Variable.

runs of the system, and especially combinations of test case generation and random testing with predictive runtime analysis would almost certainly increase the coverage of predictive runtime analysis and is worth exploring in depth, our explicit purpose in this paper is to present and evaluate predictive runtime analysis based on sliced causality *in isolation*. Careful inspection of the evaluation results revealed that the known bugs that were missed by jPredictor were missed simply because of limited test inputs: their corresponding program points were not touched during the execution. Any dynamic analysis technique suffers from this problem. Our empirical evaluation of jPredictor indicates that the use of sliced causality in predictive runtime analysis makes it less important to generate "bad" thread interleavings in order to find concurrent bugs, but more important to generate test inputs with better code coverage.

7.2.2 Race Detection

The results of race detection are shown in Table 7.2. The second column gives the number of shared variables checked in the analysis, which is in some cases smaller than the number of shared variables in Table 7.1 for the following reasons. Some shared variables were introduced by the test drivers and therefore not needed to check. Also, as already mentioned, many shared variables are just different elements of the same array and it is usually redundant to check all of them. jPredictor provides options to turn on an automatic filter that removes undesired shared variables (using class names) or randomly picks only one element in each array to check. This filter was kept on during our experiments, resulting in fewer shared variables to check. The third and the fourth columns report the size of the trace (i.e., the number of events) logged at runtime and the size of the trace constructed after preprocessing, respectively. The difference between these shows that, with the help of static analysis, the number of events to log at runtime is indeed reduced, implying a reduction of

runtime overhead.

Columns 5 to 8 show the times used in different stages of the race detection. Because JPREDICTOR needs to repeat the trace slicing, the *VC* calculation, and the property checking for every shared variable, the times shown in Table 7.2 for these three stages are the average times for one shared variable. Considering the analysis process is entirely automatic, the performance is quite reasonable. Among all the four stages, the trace slicing is the slowest, because it is performed on the complete trace. In spite of its highest algorithmic complexity, the actual race detection is the fastest part of the process. This is not unexpected though, since it works on the sliced trace containing only the property events, which is much shorter than the complete one.

The last section of Table 7.2 reports the number of races detected in our experiments. The races are categorized into three classes: harmful, benign (do not cause real errors in the system) and false (not real races). JPREDICTORreported *no false alarms* and, for all the examples used in other works except for the FTP server, e.g., hedc and Pool 1.2, it *found all the previously known dataraces.* Note that we only count the races on the same field once, so our numbers in Table 2 may appear to be smaller than those in other approaches that use the number of unsafe access pairs. Some races in the FTP server reported in [70] were missed by JPREDICTOR because the provided test driver is comparatively simple and preformed limited testing of the server, avoiding the execution of the buggy code.

Surprisingly, JPREDICTOR found some races in Pool 1.2 that were missed by the static race detector in [70], which is expected to have a very comprehensive coverage of the code (at the expense of false alarms). JPREDICTOR also reported some unknown harmful races in StaticBucketMap, Pool 1.3 and Tomcat. The race in StaticBucketMap is caused by unprotected accesses to the internal nodes of the map via the *Map.Entry* interface. It leads to a harmful atomicity violation, explained in more detail in the next subsection. Although Pool 1.3 fixed all the races found in Pool 1.2, JPREDICTOR still detected a race when an object pool is closed: in GenericObjectPool, a concrete subclasses of the abstract BaseObjectPool class, the close process first invokes the close function in the super class *without* proper synchronization. Hence, other methods can interfere with the close function, leading to unexpected exceptions.

For Tomcat, JPREDICTOR found four dataraces: two of them are benign and the other two are real bugs. Our investigation showed that they have been previously submitted to the bug database of Tomcat by other users. Both bugs are hard to reproduce and only rarely occur, under very heavy workloads; JPREDICTOR was able to catch them using only a few working threads. More

```
if ((entry == null) || (entry.binaryContent == null)
   && (entry.loadedClass == null))
     throw new ClassNotFoundException(name);

Class clazz = entry.loadedClass;
if (clazz != null)  return clazz;
```

Figure 7.3: Buggy code in WebappClassLoader

```
if (entry == null)
    throw new ClassNotFoundException(name);
Class clazz = entry.loadedClass;
if (clazz != null)  return clazz;
synchronized (this) {
if (entry.binaryContent == null && entry.loadedClass == null)
   throw new ClassNotFoundException(name);
}
```

Figure 7.4: Patched code in WebappClassLoader

interestingly, one bug was claimed to be fixed, but when we tried the patched version, the bug was still there. Let us take a close look at this bug.

This bug resides in *findClassInternal* of org.apache.catalina.loader. WebappClassLoader. This bug was first reported by jPredictor as dataraces on variables *entry.binaryContent* and *entry.loadedClass* at the first conditional statement in Figure 7.3. The race on *entry.loadedClass* does not lead to any errors, and the one on *entry.binaryContent* does no harm by itself, but *together* they may cause some arguments of a later call to *definePackage(packageName, entry.manifest, entry.codeBase)*[4] to be null, which is illegal. It seems that a Tomcat developer tried to fix this bug by putting a lock around the conditional statement, as shown in Figure 7.4. However, jPredictor showed that the error still exists in the patched code, which was a part of the latest version of Tomcat 5 when we carried out our experiments. We reported the bug with a fix and it has been accepted by the Tomcat developers.

7.2.3 Atomicity Violation Detection

The results of evaluating jPredictor on atomicity analysis are shown in Table 7.3. Although jPredictor allows the user to define different kinds of atomic blocks, we only checked for the atomicity of methods in these experiments. Not all benchmarks were checked: we do not have

[4]There is another *definePackage* function with eight arguments that allows null arguments.

Program	Running Time (seconds)			Violations	
	Slice	VC	Verify	Actual	False
Banking	0.01	0.01	0.01	1	0
Elevator	0.4	3.2	0.6	0	0
tsp	0.5	2.5	0.6	1	0
sor	0.1	0.6	0.46	0	0
hedc	0.02	0.18	0.02	1	0
StringBuffer	0.01	0.05	0.01	1	0
Vector	0.06	0.15	0.06	4	0
StaticBucketMap	8	14	0.03	1	0
Pool 1.2	0.23	1.87	3.4	10	0
Pool 1.3	0.19	0.61	0.03	0	0

Table 7.3: Atomicity analysis results

enough knowledge of the IBM Web Crawler to judge atomicity (its source code is not public), while method atomicity is not significant for FTP and Tomcat, since their methods are complex and usually not atomic (finer grained atomic blocks are more desirable there, but this is beyond our purpose in this paper).

We do not need to repeat the pre-processing stage for atomicity analysis. Hence, only the times for slicing, *VC* calculation and atomicity checking are shown in columns 2 to 4 in Table 7.3. As discussed in Section 7.1.4, our evaluation of atomicity analysis reused the trace slices generated for race detection to reduce the slicing cost, which turned out to be effective according to the results. The other two stages took more time in atomicity analysis than in race detection because the analyzed trace slice was larger. The last part of Table 7.3 shows the number of detected atomicity violations, which are divided into two categories: actual violations and false alarms. No benign violations were found in our evaluation, probably because the definition of atomicity that we adopted is based on problematic patterns of event sequences.

jPredictor *did not report any atomicity false alarm* in its analysis. It also *found all the previously known harmful atomicity violations* in the examples also analyzed by other approaches, e.g., [89] and [43]. Moreover, jPredictor found harmful atomicity violations in tsp and hedc that were missed by [89] and [43] using the same test drivers. This indicates that jPredictor, through its combination of static dependence analysis and sliced causality, provides a better capability of predicting atomicity violations. Some unknown violations in StaticBucketMap and Pool 1.2 were also detected. We next briefly explain the violation in StaticBucketMap.

In StaticBucketMap, fine grained internal locks are used to provide thread-safe map operations. Specifically, every bucket in the map is protected by a designated lock. A data race was still detected

```
StaticBucketMap map;
...
Map.Entry entry = (Map.Entry)map.entrySet().iterator().next();
entry.setValue(null);
```

Figure 7.5: Unprotected modification of the map entry

```
class MapPrinter implements Runnable{
  public void run(){
    Iterator it = map.entrySet().iterator();
    while (it.hasNext()) {
       Map.Entry entry = (Map.Entry)it.next();
       if (entry.getValue() != null)
         System.out.println(entry.getValue().toString());
    }
  }
  public void atomicPrint(){
    map.atomic(this);
  }
}
```

Figure 7.6: Atomic iteration on the map

by jPredictor in this well synchronized implementation, caused by the usage of the *Map.Entry* interface. As shown in Figure 7.5, one can obtain a map entry, which represents a key-value pair, via an iterator of the map and use the *setValue* method to change the entry. jPredictor showed that the *setValue* method is not correctly synchronized and causes a data race. This data race is benign in most cases, because no new entry can be added or removed through the *Map.Entry* interface and also because the bulk operations of the map, e.g., iteration, are not guaranteed to be atomic. However, StaticBucketMap provides an *atomic(Runnable r)* method to support atomic bulk operations. This method accepts a *Runnable* object and executes the *run()* method of the object atomically with regards to the map. Figure 7.6 shows an example of using this method to print out all the values in the map atomically. However, this atomicity guarantee can be violated when another thread accesses the map's elements using the unsafe *setValue* method, like the code in Figure 7.6, which can cause an unexpected null pointer exception. jPredictor detected this violation (without directly hitting it during the execution) in our experiments, generating a warning message that clearly points out the cause of the violation.

Chapter 8

Related Work

Numerous approaches have been introduced based on runtime monitoring and analysis. Here we mainly focus on those related to runtime verification and concurrent program analayis.

8.1 Runtime Monitoring Related

We next discuss relationships between MOP and other related paradigms, including AOP, design by contract, runtime verification, and other trace monitoring approaches. Broadly speaking, all the monitoring approaches discussed below are runtime verification approaches; however, in this section only, we group into the runtime verification category only those approaches that explicitly call themselves runtime verification approaches. Interestingly, even though most of the systems mentioned below target the same programming languages, no two of them share the same logical formalism for expressing properties. This observation strengthens our belief that probably there is *no silver bullet logic* (or *super logic*) for all purposes. A major objective in the design of MOP was to avoid hardwiring particular logical formalisms into the system. In fact, as shown in Section 3.3, MOP specifications are generic in four orthogonal directions:

MOP[logic, scope, running mode, handlers].

The logic answers *how to specify* the property. The scope answers *where to check* the property; it can be class invariant, global, interface, etc. The running mode answers *where the monitor is*; it can be inline, online, offline. The handlers answer *what to do if*; there can be violation and validation handlers. For example, a particular instance can be

MOP[ERE, global, inline, validation],

where the property is expressed using the ERE logic-plugin for extended regular expressions (EREs), the corresponding monitor is global and inline, and validation of the formula (pattern matching in this case) is of interest.

Most approaches below can be seen as such specialized instances of MOP for particular logics, scopes, running modes and handlers. There are, of course, details that make each of these approaches interesting in its own way.

8.1.1 Aspect Oriented Programming (AOP) Languages

Since its proposal in [59], AOP has been increasingly adopted and many tools have been developed to support AOP in different programming languages, e.g., AspectJ and JBoss [54] for Java and AspectC++ [7] for C++. Built on these general AOP languages, numerous extensions have been proposed to provide domain-specific features for AOP. Among these extensions, Tracematches [3] and J-LO [20] support history(trace)-based aspects for Java.

Tracematches enables the programmer to trigger the execution of certain code by specifying a regular pattern of events in a computation trace, where the events are defined over entry/exit of AspectJ pointcuts. When the pattern is matched during the execution, the associated code will be executed. In this sense, Tracematches supports trace-based pointcuts for AspectJ. J-LO is a tool for runtime-checking temporal assertions. These temporal assertions are specified using LTL and the syntax adopted in J-LO is similar to Tracematches' except that the formulae are written in a different logic. J-LO mainly focuses on checking at runtime properties rather than providing programming support. In J-LO, the temporal assertions are inserted into Java files as annotations that are then compiled into runtime checks. Both Tracematches and J-LO support parametric events, i.e., free variables can be used in the event patterns and will be bound to specific values at runtime for matching events. Conceptually, J-LO can be captured by MOP, because LTL is supported by MOP and J-LO's temporal assertions can be easily translated into MOP specifications that contain only action events and validation handlers.

Both Tracematches and J-LO hardwire logics for specifying properties, making them less expressive than MOP. Also, as shown in Section 5.3, the Java instance of MOP, JavaMOP, generates more efficient monitoring code than Tracematches. It is also worth mentioning that Tracematches and J-LO are implemented using Java bytecode compilation and instrumentation, while MOP acts as an aspect synthesizer, making it easier to port to other target languages provided they have AOP tool support.

8.1.2 Runtime Verification

In runtime verification, monitors are automatically synthesized from formal specifications, and can be deployed *offline* for debugging, or *online* for dynamically checking properties during execution. MaC [60], PathExplorer (PaX) [47], and Eagle [15] are runtime verification frameworks for logic based monitoring, within which specific tools for Java – Java-MaC, Java PathExplorer, and Hawk [34], respectively – are implemented. All these runtime verification systems work in outline monitoring mode and have hardwired specification languages: MaC uses a specialized language based on interval temporal logic, JPaX supports just LTL, and Eagle adopts a fixed-point logic. Java-Mac and Java PathExplorer integrate monitors via Java bytecode instrumentation, making them difficult to port to other languages. Our approach supports inline, outline and offline monitoring, allows one to define new formalisms to extend the MOP framework, and is adaptable to new programming languages.

Temporal Rover [35] is a commercial runtime verification tool based on future time metric temporal logic. It allows programmers to insert formal specifications in programs via annotations, from which monitors are generated. An Automatic Test Generation (ATG) component is also provided to generate test sequences from logic specifications. Temporal Rover and its successor, DB Rover, support both inline and offline monitoring. However, they also have their specification formalisms hardwired and are tightly bound to Java.

Although our current JavaMOP prototype does not support all these techniques yet, it is expected that all the RV systems would fall under the general MOP architecture, provided that appropriate logic-plugins are defined.

8.1.3 Design by Contract

Design by Contract (DBC) [68] is a technique allowing one to add semantic specifications to a program in the form of assertions and invariants, which are then compiled into runtime checks. It was first introduced by Meyer as a built-in feature of the Eiffel language [38]. Some DBC extensions have also been proposed for a number of other languages. Jass [17] and jContractor [1] are two Java-based approaches.

Jass is a precompiler which turns the assertion comments into Java code. Besides the standard DBC features such as pre-/post- conditions and class invariants, it also provides refinement checks. The design of trace assertions in Jass is mainly influenced by CSP [49], and the syntax is more like a programming language. jContractor is implemented as a Java library which allows programmers to associate contracts with any Java classes or interfaces. Contract methods can be included directly

within the Java class or written as a separate contract class. Before loading each class, jContractor detects the presence of contract code patterns in the Java class bytecode and performs on-the-fly bytecode instrumentation to enable checking of contracts during the program's execution. jContractor also provides a support library for writing expressions using predicate logic quantifiers and operators such as *Forall*, *Exists*, *suchThat*, and *implies*. Using jContractor, the contracts can be directly inserted into the Java bytecode even without the source code.

Java modeling language (JML)[64] is a behavioral interface specification language for Java. It provides a more comprehensive modeling language than DBC extensions. Not all features of JML can be checked at runtime; its runtime checker supports a DBC-like subset of JML, a large part of which is also supported by JavaMOP. Spec# [14] is a DBC-like extension of the object-oriented language C#. It extends the type system to include non-null types and checked exceptions and also provides method contracts in the form of pre- and post-conditions as well as object invariants. Using the Spec# compiler, one can statically enforce non-null types, emit run-time checks for method contracts and invariants, and record the contracts as metadata for consumption by downstream tools.

We believe that the logics of assertions/invariants used in DBC approaches fall under the uniform format of our logic engines, so that an MOP environment following our principles would naturally support monitoring DBC specifications as a special methodological case. In addition, our MOP design also supports outline monitoring, which we find important in assuring software reliability but which is not provided by any of the current DBC approaches that we are aware of.

8.1.4 Other Related Approaches

Acceptability-oriented computing [75] aims at enhancing flawed computer systems to respect basic acceptability properties. For example, by augmenting the compiled code with bounds checks to detect and discard out-of-bound memory accesses, the system may execute successfully through attacks that trigger otherwise fatal memory errors. Acceptability-oriented computing is mainly a philosophy and methodology for software development; one has to devise specific solutions to deal with different kinds of failures. We do believe though that MOP can serve as a platform to experiment with and support acceptability-oriented computing, provided that appropriate specification formalisms express the "acceptability policy" and appropriate recovery ensures that it is never violated.

Program Query Language (PQL) allows programmers to express design rules that deal with sequences of events associated with a set of related objects [66]. Both static and dynamic tools have been implemented to find solutions to PQL queries. The static analysis conservatively looks for

potential matches for queries and is useful to reduce the number of dynamic checks. The dynamic analyzer checks the runtime behavior and can perform user-defined actions when matches are found, similar to MOP handlers.

PQL has a "hardwired" specification language based on context-free grammars (CFG) and supports only inline monitoring. CFGs can potentially express more complex languages than regular expressions, so in principle PQL can express more complex safety policies than Tracematches. There is an unavoidable trade-off between the generality of a logic and the efficiency of its monitors; experiments performed by Tracematches colleagues [11] and confirmed by us (see Section 7.2) show that PQL adds, on average, more than twice as much runtime overhead as Tracematches. We intend to soon take a standard CFG-to-pushdown-automata algorithm and to implement it as an MOP logic-plugin; then MOP will also support (the rare) CFG specifications that cannot be expressed using parametric extended regular expressions or temporal logics, and MOP[CFG,global,inline,validation] will provide an alternative and more general implementation of PQL.

Program Trace Query Language (PTQL) [44] is a language based on SQL-like relational queries over program traces. The current PTQL compiler, Partiqle, instruments Java programs to execute the relational queries on the fly. PTQL events are timestamped and the timestamps can be explicitly used in queries. PTQL queries can be arbitrary complex and, as shown in [44], PTQL's runtime overhead seems acceptable in many cases but we were unable to obtain a working package of PTQL and compare it in our experiments because of license issues. PTQL properties are globally scoped and their running mode is inline. PTQL provides no support for recovery, its main use being to detect errors. It would be interesting to investigate the possibility of developing an SQL logic-plugin for MOP and then to compare the corresponding MOP instance to Partiqle.

8.2 Concurrent Program Analysis

There are several other approaches also aiming at detecting potential concurrency errors by examining particular execution traces. Some of these approaches aim at verifying general purpose properties [81, 82], including temporal ones, and are inspired from debugging distributed systems based on Lamport's happens-before causality [63]. Other approaches work with particular properties, such as data-races and/or atomicity. [80] introduces a first lock-set based algorithm to detect data-races dynamically, followed by many variants aiming at improving its accuracy. For example, an ownership model was used in [88] to achieve a more precise race detection at the object level.

[72] combines the lock-set and the happen-before techniques. The lock-set technique has also been used to detect atomicity violations at runtime, e.g., the reduction based algorithms in [43] and [89]. [89] also proposes a block-based algorithm for dynamic checking of atomicity built on a simplified happen-before relation, as well as a graph-based algorithm to improve the efficiency and precision of runtime atomicity analysis.

Previous efforts tend to focus on either soundness or coverage: those based on happens-before try to be sound, but have limited coverage over interleavings, resulting in more false negatives; lock-set based approaches have better coverage but suffer from false alarms. Our runtime analysis technique proposed in this thesis aims at covering more interleavings from one execution without giving up soundness or genericity of properties (errors may still be missed, e.g., when the code causing errors is not executed in one execution).

Chapter 9

Conclusion

This thesis presents runtime monitoring-based approaches to detect and react to system errors on the fly. We first introduce monitoring oriented programming, which is a generic, efficient runtime verification framework. MOP automatically synthesizes efficient monitoring code from user-specified properties and integrates the monitoring code into the program to monitor. The logic that one can use to specify properties in MOP is extensible: a new logic can be easily plugged into MOP through an MOP logic plugin that encapsulates a corresponding monitor generation algorithm. MOP also provides logic-independent support for monitoring of parametric properties, allowing one to re-use any monitoring algorithm for non-parametric specifications to support parametric specifications without much effort. We have developed an MOP instance for the Java language, named JavaMOP. JavaMOP has been applied to a large number of applications and properties. It generates more efficient monitoring code than any other existing monitoring techniques and helped us to find tricky bugs in popular programs.

MOP provides an effective solution to finding semantics-related errors from observed executions. However, it suffers from the same limited coverage as testing, especially for concurrent programs. We thus developed predictive runtime analysis, which increases the coverage of runtime analysis and predicts concurrent errors even when they have not occured. In predictive runtime analysis, causal partial orders are extracted from an observed executions and then used to infer potential executions by generating their consistent linearizations. Hence, the restrictiveness of the computed causal partial orders determines the prediction capabilty. We have devised sliced causality, which rastically cuts the usual partial order set extracted by the happen-before causality by removing unnecessary dependencies, resulting better capability of detecting potential errors. The predictive runtime analysis technique has been implemented in jPredictor, a tool to detect concurrent errors in Java programs. The evaluation results show that jPredictor is effective and accurate in finding concurrent errors.

References

[1] P. Abercrombie and M. Karaorman. jContractor: Bytecode instrumentation techniques for implementing DBC in Java. In *Runtime Verification*, volume 70.4 of *ENTCS*, 2002.

[2] Hiralal Agrawal and Joseph Robert Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90)*, 1990.

[3] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*, pages 345–364. ACM, 2005.

[4] Chris Anley. Advanced SQL injection in SQL server applications. *NGSSoftware*, 2002.

[5] Apache Commons project. http://commons.apache.org/.

[6] Apache FTP server project. incubator.apache.org/ftpserver/.

[7] AspectC++. http://www.aspectc.org/.

[8] C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Rosu, and W. Visser. Experiments with test case generation and runtime analysis. In *ASM'03*, volume 2589 of *LNCS*, pages 87–107, 2003.

[9] AspectJ. http://eclipse.org/aspectj/.

[10] AspectJ language. http://www.eclipse.org/aspectj/doc/released/progguide/language.html.

[11] Pavel Avgustinov, Eric Bodden, Elnar Hajiyev, Laurie Hendren, Ondrej Lhotak, Oege de Moor, Neil Ongkingco, Damien Sereni, Ganesh Sittampalam, Julian Tibble, and Mathieu Verbaere. Aspects for trace monitoring. In *FATES/RV'06*, volume 4262 of *LNCS*, pages 20–39, 2006.

[12] Pavel Avgustinov, Julian Tibble, Eric Bodden, Ondrej Lhotak, Laurie Hendren, Oege de Moor, Neil Ongkingco, and Ganesh Sittampalam. Efficient trace monitoring. Technical Report abc-2006-1, Oxford University, 2006.

[13] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making trace monitors feasible. In *OOPSLA'07*, pages 589–608. ACM, 2007.

[14] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS'04*, volume 3362 of *LNCS*, pages 49–69, 2004.

[15] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-Based Runtime Verification. In *VMCAI'04*, volume 2937 of *LNCS*, pages 44–57, 2004.

[16] Howard Barringer, Bernd Finkbeiner, Yuri Gurevich, and Henny Sipma. *Runtime Verification (RV'05)*. Elsevier, 2005. ENTCS 144.

[17] Detlef Bartetzko, Clemens Fischer, Michael Moller, and Heike Wehrheim. Jass-Java with Assertions. In *Runtime Verification*, volume 55.2 of *ENTCS*, 2001.

[18] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06*, pages 169–190. ACM, 2006.

[19] Bnf definition. http://en.wikipedia.org/wiki/Backus-Naur_form.

[20] Eric Bodden. J-LO, a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, 2005.

[21] Eric Bodden, Feng Chen, and Grigore Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *Aspect-Oriented Software Development (AOSD'09)*, pages 3–14. ACM, 2009.

[22] Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP'07*, volume 4609 of *LNCS*, pages 525–549, 2007.

[23] Graham Brightwell and Peter Winkler. Counting linear extensions is #p-complete. In *ACM symposium on Theory of computing (STOC'91)*, 1991.

[24] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[25] Feng Chen, Marcelo D'Amorim, and Grigore Roşu. Checking and correcting behaviors of Java programs at runtime with JavaMOP. In *Runtime Verification(RV'06)*, volume 144 (4) of *ENTCS*, pages 3–20, 2006.

[26] Feng Chen and Grigore Roşu. JavaMOP. http://fsl.cs.uiuc.edu/javamop.

[27] Feng Chen and Grigore Roşu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Runtime Verification(RV'03)*, volume 89(2) of *ENTCS*, 2003.

[28] Feng Chen and Grigore Roşu. Predicting concurrency errors at runtime using sliced causality. Technical Report UIUCDCS-R-2005-2660, Department of CS at UIUC, 2005.

[29] Feng Chen and Grigore Roşu. MOP: An efficient and generic runtime verification framework. In *OOPSLA'07*, pages 569–588. ACM, 2007.

[30] Feng Chen and Grigore Roşu. Parametric and sliced causality. In *CAV'07*, volume 4590 of *LNCS*, pages 240–253, 2007.

[31] Feng Chen and Grigore Roşu. Parametric trace slicing and monitoring. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261, 2009.

[32] Feng Chen and Grigore Roşu. Parametric and termination-sensitive control dependence - extended abstract. In *Static Analysis Symposium (SAS'06)*, 2006.

[33] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In *ACM/ONR workshop on Parallel and distributed debugging (PADD)*, 1991.

[34] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of Java programs. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[35] D. Drusinsky. Temporal Rover, 1997–2009. http://www.time-rover.com.

[36] Eclipse. http://eclipse.org.

[37] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[38] Eiffel Language. http://www.eiffel.com/.

[39] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.

[40] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal analysis of java programs in javafan. In *Proceedings of Computer-aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 501 – 505, 2004.

[41] Azadeh Farzan and Madhusudan Parthasarathy. Causal atomicity. In *Computer Aided Verification (CAV'06)*, 2006.

[42] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[43] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Principles of Programming Languages (POPL'04)*, 2004.

[44] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA'05*, pages 385–402. ACM, 2005.

[45] Tibor Gyimothy, Arpad Beszedes, and Istvan Forgacs. An efficient relevant slicing method for debugging. In *ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, 1999.

[46] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Static Analysis Symposium (SAS'99)*, 1999.

[47] Klaus Havelund and Grigore Roşu. Monitoring Java programs with Java PathExplorer. In *Runtime Verification(RV'01)*, volume 55(2) of *ENTCS*, 2001.

[48] Klaus Havelund and Grigore Roşu. *Runtime Verification (RV'01, RV'02, RV'04)*. Elsevier, 2001, 2002, 2004. ENTCS 55, 70, 113.

[49] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall Intl., New York, 1985.

[50] Hendrick Jan Hoogeboom and Grzergorz Rozenberg. Dependence graphs. In Volker Diekert and Grzegorz Rozenberg, editors, *The Book of Traces*, pages 43–67. World Scientific, 1995.

[51] Susan Horwitz and Thomas W. Reps. The use of program dependence graphs in software engineering. In *International Conference on Software Engineering (ICSE'92)*, 1992.

[52] Java. http://java.sun.com.

[53] Java specification. http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html.

[54] JBoss. http://www.jboss.org.

[55] jHotdraw. http://www.jhotdraw.org.

[56] Selmer M. Johnson. Generation of permutations by adjacent transposition. *Mathematics of Computation*, 17(83):282–285, 1963.

[57] jPredictor. http://fsl.cs.uiuc.edu/jPredictor/.

[58] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP'01*, volume 2072 of *LNCS*, pages 327–353, 2001.

[59] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, volume 1241 of *LNCS*, pages 220–242, 1997.

[60] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a runtime assurance tool for Java. In *Runtime Verification*, 2001.

[61] MoonZoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[62] Jens Krinke. Static slicing of threaded programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 1998.

[63] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of ACM*, 21(7):558–565, 1978.

[64] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA'00*, 2000.

[65] Lucene. http://lucene.apache.org.

[66] Michael Martin, V. Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA'07*, pages 365–383. ACM, 2005.

[67] Patrick Meredith, Dongyun Jin, Feng Chen, and Grigore Roşu. Efficient monitoring of parametric context-free patterns. In *Automated Software Engineering (ASE '08)*, pages 148–157. IEEE, 2008.

[68] B. Meyer. *Object-Oriented Software Construction,* 2^{nd} *edition*. Prentice Hall, New Jersey, 2000.

[69] Edward F. Moore. Gedanken-experiments on sequential machines. *Automata Studies, Annals of Math. Studies*, 34:129–153, 1956.

[70] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. *ACM SIGPLAN conference on Programming language design and implementation (PLDI'06)*, 2006.

[71] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 1991.

[72] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, 2003.

[73] Rodolfo Pellizzoni, Patrick Meredith, Marco Caccamo, and Grigore Roşu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time System Symposium (RTSS'08)*, pages 481–491. IEEE, 2008.

[74] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.

[75] M. Rinard. Acceptability-oriented computing. In *Onward! Track, OOPSLA'03*, 2003.

[76] G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Journal of ASE*, 12(2), 2005.

[77] Grigore Roşu. An effective algorithm for the membership problem for extended regular expressions. In *Proceedings of the Tenth International Conference on Foundations Of Software Science and Computation Structures (FOSSACS'07)*, 2007.

[78] Grigore Roşu and Klaus Havelund. Rewriting-based techniques for runtime verification. *J. Automated Software Engineering*, 12(2):151–197, 2005.

[79] Grigore Roşu, Feng Chen, and Thomas Ball. Synthesizing monitors for safety properties – this time with calls and returns –. In *Workshop on Runtime Verification (RV'08)*, volume 5289 of *Lecture Notes in Computer Science*, pages 51–68. Springer, 2008.

[80] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transaction of Computer System*, 15(4):391–411, 1997.

[81] Alper Sen and Vijay K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In *International Conference on Principles of Distributed Systems (OPODIS'03)*, 2003.

[82] Koushik Sen, Grigore Roşu, and Gul Agha. Runtime safety analysis of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'03)*, 2003.

[83] Koushik Sen, Grigore Roşu, and Gul Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, 2005.

[84] O. Sokolsky and M. Viswanathan. *Runtime Verification (RV'03)*. Elsevier, 2003. ENTCS 89.

[85] Soot website. http://www.sable.mcgill.ca/soot/.

[86] Apache group. Tomcat. http://jakarta.apache.org/tomcat/.

[87] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Principles of Programming Languages (POPL'06)*, 2006.

[88] Christoph von Praun and Thomas R. Gross. Object race detection. In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, 2001.

[89] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, 2006.

[90] Xalan. http://xml.apache.org/xalan-j/.

Vita

Feng Chen was born in Long-Hai, Fujian Province, China, on April 3rd, 1977.

Research Interests

Automated software engineering and software quality, safety and security. Programming and specification language semantics, design and implementation. Dynamic and static program analysis and verification. Applications of formal methods and algorithm design.

Education

Ph.D. Computer Science, University of Illinois at Urbana-Champaign, USA, 2009 (expected)
Dissertation: "Monitoring Oriented Programming and Analysis".
Adviser: Professor Grigore Rosu

M.S. Computer Science, Peking University, China, 2002
Adviser: Professor Fuqing Yang

B.A. Computer Science, Peking University, China, 1999

Awards and Distinctions

Reverse chronological order:

- *ACM SIGSOFT Distinguished Paper Award*, for [ASE 2008].
- *C.L. and Jane Liu Award*, Department of Computer Science, UIUC, 2005. Prize offered once a year to a most promising graduate student.
- *Wu Si Award*, Peking University, 2002. Prize offered once a year to best graduate students.
- *Huawei Fellowship*, Peking University, 1998.

- *Xiyue Fellowship*, Peking University, 1997.
- *Excellent Student Awards*, Peking University, 1996, 1997 and 1998. Prize offered once a year to outstanding undergraduate students.

Professional and Academic Experience

Research

- Fall 2002 to present. **Research assistant**. Formal System Laboratory, Department of Computer Science, UIUC. Adviser: Grigore Rosu.

  Involved in several NSF/NASA funded projects. Developed *monitoring oriented programming* (MOP), a generic and efficient framework for runtime monitoring and recovery. Several instances of MOP have implemented for different applications. Evaluation showed that MOP generates more efficient monitoring code than other specialized approaches. Defined and implemented *sliced causality*, which drastically but soundly slices Lamport's happen-before causality using static analysis, and *predictive runtime analysis* (PRA), an effective technique to predict potential concurrency errors from observed executions. Implemented a PRA tool for Java, named jPredictor, which detected many unknown bugs in popular open source systems. Formalized the semantics of Java 1.4 in rewrite logic and implemented JavaRL, a static formal analysis framework for multithreaded Java programs. Co-implemented a static pluggable domain-specific policy checker for C based on a rewrite logic semantic definition designed for the symbolic execution of C.

- Summer 2005. **Research intern**. Foundations of Software Engineering Group at Microsoft Research. Mentors: Wolfram Schulte and Nikolai Tillmann.

  Developed Axiom Meister, a tool to automatically infer interface contracts for .Net programs based on symbolic execution. It generates concise, human friendly and comprehensive specifications. Found design flaws of libraries.

- Fall 1999 to Summer 2002. **Research assistant**. Software Engineering Group, Department of Computer Science, Peking University, China. Adviser: Fuqing Yang.

  Involved in several nation key research programs. Developed ABC, a tool-supported, architecture-based approach for component oriented software development. Implemented the core part of PKUAS, a high performance, adaptive application server.

- Summer 1999. **Research intern**. Bell Labs Research China. Mentor: Keqing Li.

  Developed STAR-RE, a flexible Telecom maintenance management system.

Teaching

- Fall 2006, Fall 2007, Fall 2008. **Guest Lecturer** for CS476 "Program Verification". Department of Computer Science, UIUC.
- Fall 2007. **Guest Lecturer** for CS422 "Programming Language Design" and CS476 "Program Verification". Department of Computer Science, UIUC.
- Fall 2006. **Teaching assistant** for CS102 "Introduction to Computing for Non-Techical Majors". Department of Computer Science, UIUC. Lectured three one-hour sections per week.
- Fall 2001. **Teaching assistant** for "Advanced Software Engineering". Department of Computer Science, Peking University, China.

Publications

All papers are available on line at `http://fsl.cs.uiuc.edu/~fengchen`.

ASE 2009. "Efficient Formalism-Independent Monitoring of Parametric Properties", Feng Chen, Dongyun Jin, Patrick Meredith, and Grigore Roşu. Proceedings of *the 24rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Auckland, New Zealand, November 16 - 20, 2008. ACM Press, to appear, 2009. (Acceptance rate: 17%)

TACAS 2009. "Parametric Trace Slicing and Monitoring", Feng Chen and Grigore Roşu. Proceedings of *the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, York, UK, March 22 - 29, 2009. To appear. (Acceptance rate: 20%)

AOSD 2009. "Dependent advice: A general approach to optimizing history-based aspects", Eric Bodden, Feng Chen, and Grigore Roşu. Proceedings of *the 8th International Conference on Aspect-Oriented Software Development (AOSD)*, Charlottesville, Virginia, March 2 - 6, 2009. (Acceptance rate: 28%)

ASE 2008. "Efficient Monitoring of Parametric Context-Free Patterns", Patrick Meredith, Dongyun Jin, Feng Chen, and Grigore Roşu. Proceedings of *the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, L'Aquila, Italy, September 15 - 19, 2008. ACM Press, pages 148-157. **ACM Distinguished Paper Award**. (Acceptance rate: 12%)

RULE 2008. "A Rewriting Logic Approach to Static Checking of Units of Measurement in C", Mark Hills, Feng Chen, and Grigore Roşu. Proceedings of *the 9th International Workshop on Rule-Based Programming (RULE)*, Hagenberg, Austria, July 18, 2008. Electronic Notes in Theoretical Computer Science, to appear.

ICSE 2008. "jPredictor: A Predictive Runtime Analysis Tool for Java", Feng Chen, Traian Florin Şerbănuţă, and Grigore Roşu. Proceedings of *the 30th International Conference on Software Engineering (ICSE)*, Leipzig, Germany, May 10 - 18, 2008. ACM Press, pages 221-230. (Acceptance rate: 15%)

RV 2008. "Synthesizing Monitors for Safety Properties – This Time With Calls and Returns", Grigore Roşu, Feng Chen, and Thomas Ball. Proceedings of *the 8th International Workshop on Runtime Verification (RV)*, Budapest, Hungary, March 30, 2008. Lecture Notes in Computer Science, Volume 5289, pages 51-68. (Acceptance rate: 33%)

OOPSLA 2007. "MOP: An Efficient and Generic Runtime Verification Framework", Feng Chen and Grigore Roşu. Proceedings of *the 22nd ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Montreal, Quebec, Canada, October 21 - 25, 2007. ACM Press, pages 569-588. (Acceptance rate: 21%)

CAV 2007. "Parametric and Sliced Causality", Feng Chen and Grigore Roşu. Proceedings of *the 19th International Conference on Computer Aided Verification (CAV)*, Berlin, Germany, July 3-7, 2007. Lecture Notes in Computer Science, Volume 4590, pages 240-253. (Acceptance rate: 24%)

ICFEM 2006. "Discovering Likely Method Specifications", Nikolai Tillmann, Feng Chen, and Wolfram Schulte. Proceedings of *the 8th International Conference on Formal Engineering Methods (ICFEM)*, Macao, China, November 1-3, 2006. Lecture Notes in Computer Science, Volume 4260, pages 717-736. (Acceptance rate: 35%)

SAS 2006. "Parametric and Termination-Sensitive Control Dependence", Feng Chen and Grigore Roşu. Proceedings of *the 13th International Static Analysis Symposium (SAS)*, Seoul, Ko-

rea, August 29-31, 2006. Lecture Notes in Computer Science, Volume 4134, pages 387-404. (Acceptance rate: 28%)

RV 2005. "Checking and Correcting Behaviors of Java Programs at Runtime with Java-MOP", Feng Chen and Grigore Roşu. Proceedings of *the 5th Workshop on Runtime Verification (RV)*, Edinburgh, UK, July 12, 2005. Electronic Notes in Theoretical Computer Science, Volume 144, pages 3-20.

TACAS 2005. "Java-MOP: A Monitoring Oriented Programming Environment for Java", Feng Chen and Grigore Roşu. Proceedings of *the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, tool paper, Edinburgh, UK, April 4-8, 2005. Lecture Notes in Computer Science, Volume 3440, pages 546-550. (Acceptance rate: 24%)

ICFEM 2004. "A Formal Monitoring-based Framework for Software Development and Analysis", Feng Chen, Marcelo D'Amorim, and Grigore Roşu. Proceedings of *the 6th International Conference on Formal Engineering Methods (ICFEM)*, Seattle, USA, November 8-12, 2004. Lecture Notes in Computer Science, Volume 3308, pages 357-373. (Acceptance rate: 27%)

CAV 2004. "Formal Analysis of Java Programs in JavaFAN", Azadeh Farzan, Feng Chen, José Meseguer and Grigore Roşu. Proceedings of *the 16th International Conference on Computer Aided Verification (CAV)*, tool paper, Boston, MA, USA, July 13-17, 2004. Lecture Notes in Computer Science, Volume 3114, pages 501-505. (Acceptance rate: 22%)

ASE 2003. "Certifying Measurement Unit Safety Policy", Feng Chen and Grigore Roşu. Proceedings of *the 18th IEEE International Conference on Automated Software Engineering (ASE)*, system paper, Montreal, Quebec, Canada, October 6-10, 2003. IEEE Computer Society, pages 304-309. (Acceptance rate: 13%)

RV 2003. "Towards Monitoring-Oriented Programming:A Paradigm Combining Specification and Implementation", Feng Chen and Grigore Roşu. Proceedings of *the 3rd Workshop on Runtime Verification (RV)*, Boulder, Colorado, USA, July 13, 2003. Electronic Notes in Theoretical Computer Science, Volume 89, pages 108-127.

RTA 2003. "Rule-Based Analysis of Dimensional Safety", Feng Chen, Grigore Grigore Roşu, and Ram prasad Venkatesan. Proceedings of *14th International Conference on Rewriting Tech-*

niques and Applications(RTA), Valencia, Spain, June 9-11, 2003. Lecture Notes in Computer Science, Volume 2706, pages 197-207. (Acceptance rate: 43%)

ICFEM 2002. "ABC/ADL: An ADL Supporting Component Composition", Hong Mei, Feng Chen, Qianxiang Wang, and Yaodong Feng. Proceedings of *the 4th International Conference on Formal Engineering Methods (ICFEM)*, Shanghai, China, October 21-25, 2002. Lecture Notes in Computer Science, Volume 2459, pages 38-47. (Acceptance rate: 40%)

ICSM 2002. "Using Application Server To Support Online Evolution", Qianxiang Wang, Feng Chen, Hong Mei, and Fuqing Yang. Proceedings of *18th International Conference on Software Maintenance (ICSM)*, Montreal, Quebec, Canada, October 3-6, 2002. IEEE Computer Society, pages 131-140. (Acceptance rate: 43%)

CompSAC 2002. "An Architecture-Based Approach for Component-Oriented Development", Feng Chen, Qianxiang Wang, Hong Mei, and Fuqing Yang. Proceedings of *26th International Computer Software and Applications Conference (CompSAC)*, Oxford, UK, August 26-29, 2002. IEEE Computer Society, pages 450-455. (Acceptance rate: 38%)

Technical Reports in Submission

1) [TR 2008-3.] "Maximal Causal Models for Multithreaded Systems", Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu. No. UIUCDCS-R-2008-3017, Department of Computer Science, UIUC, 2008.

2) [TR 2008-2.] "Mining Parametric State-Based Specifications from Executions", Feng Chen, and Grigore Roşu. No. UIUCDCS-R-2008-3000, Department of Computer Science, UIUC, 2008.

Presentations

- "Runtime Monitoring for Reliable Software", seminar lecture, Department of Computer Science, Cornell University. April 15, 2008.
- "Runtime Monitoring for Reliable Software", seminar lecture, Department of Electrical and Computer Engineering, University of Texas at Austin. April 3, 2008.
- "Runtime Monitoring for Reliable Software", seminar lecture, College of Computing, Georgia Institute of Technology. March 4th, 2008.

- "Runtime Monitoring for Reliable Software", seminar lecture, Department of Computer Science, University of Texas at Arlington. February 18th, 2008.
- "Parametric and Sliced Causality", CAV 2007, Berlin, Germany. July 2007.
- "Parametric and Termination-Sensitive Control Dependence", SAS 2006, Seoul, Korea. August 2006.
- "Discovering Likely Method Specifications", with Nikolai Tillmann and Wolfram Schulte. Microsoft Research, Seattle, Washington, USA. August 2005.
- "Java-MOP: A Monitoring Oriented Programming Environment for Java", TACAS 2005, Edinburgh, UK. April 2005.
- "A Formal Monitoring-based Framework for Software Development and Analysis", ICFEM 2004, Seattle, Washington, USA. November 2004.
- "Formal Analysis of Java Programs in JavaFAN", Formal Methods seminar, Department of Computer Science, UIUC. June 2004.
- "Certifying Measurement Unit Safety Policy", ASE 2003, Montreal, Quebec, Canada. October 8, 2003.

Selected Software Systems Work

MOP. `http://fsl.cs.uiuc.edu/mop`

With Grigore Roşu at UIUC. An efficient and generic runtime verification framework.

jPredictor. `http://fsl.cs.uiuc.edu/jpredictor`

With Grigore Roşu at UIUC. A runtime predictive analysis tool to detect concurrency errors.

JavaFAN. `http://fsl.cs.uiuc.edu/javafan`

With Azadeh Farzan, Grigore Roşu and José Meseguer at UIUC. A static analysis framework for Java programs based on formal rewriting logic definitions of the Java semantics.

Professional Activities

- Organization committee for the 6th International Conference on Quality Software (QSIC'06), Beijing, China, October 27-28, 2006.

- Reviewer for 4 journals: Automated Software Engineering, IEEE Transactions on Dependable and Secure Computing, Journal of Logic and Algebraic Programming, Real-Time Systems.
- External Reviewer for 5 Conferences and Workshops: FOSSACS 2008, PLDI 2007, PADTAD 2006, RV 2006, RV 2005.